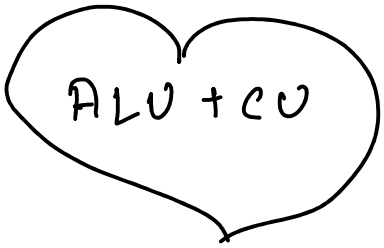
Slides

Wednesday, 1 February, 2023 8:59 PM

Slide 1

Microprocessor:

→ Single IC package
→ numerous pins
→ single chip CPU
→ Communicate through ports
→ Two types of port:
  ↳ Sepial Port
  ↳ Parallel port



→ Instruction execution

⇒ [Fetch → decode → execute → write]

↳ Basic steps of a microprocessor

Microprocessor specification:

→ A (n) bit processor has (n) bit data bus.
→ A (n) " " can address 2^n memory locations

Classification of Microprocessor:

Based on size of data bus

- 4bit
- 8bit
- 16bit
- 32 bit
- 64 bit

Based on application:

- General purpose microprocessor
- Special " "

Based on architecture:

- RISC → simle, fixed size instructions → requires memory access.
- CISC

But where is ISHQ?

RISC vs CISC

RISC	CISC
(i) Reduced number of instruction class. Simple operations that can each execute in a single cycle.	(i) Complex instructions set. Takes many cycles to complete the task.
(ii) Lage general	(ii) Specific. purpose

(ii) Large general purpose register set

(iii) Operates on data held in registers.

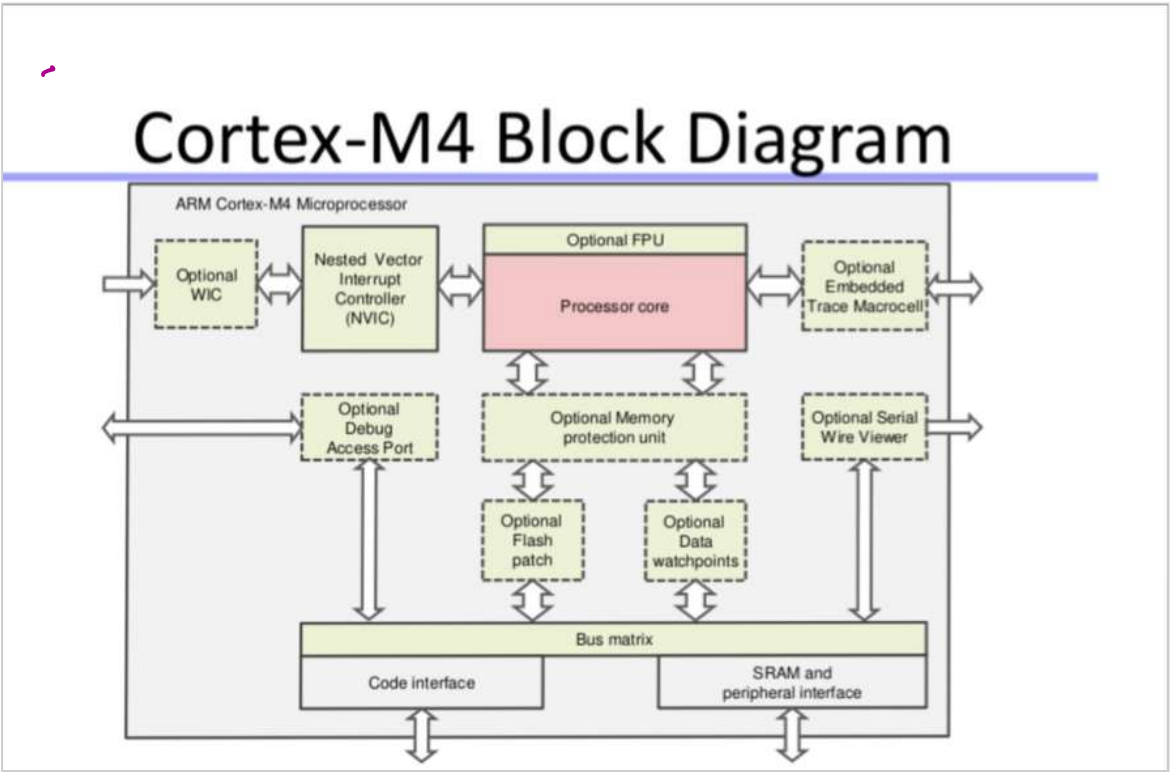
(iv)

AEBF03DB-F00F-4FE...

(ii) Specific. purpose register set.

(iii) Acts on memory directly.

Slide-2



Cortex m4 is built around the ARMv7 architecture and has 3 stage pipeline.

↳ T...

Single Pipeline

- Instruction fetch
- decode/execute
- memory access

MPU – memory protection unit

- protects memory access to specific regions of memory
- provides security.

DMA - Direct Memory Access

- allows data to be transferred between different peripherals & memory without involving the CPU.
- provides faster data transfer
- relieves the CPU overloading.

Cortex M4 Overview:

⇒ Harvard Architecture → separated data bus & instruction bus.

⇒ High performance
⇒ Low power consumption
⇒ Instruction set
⇒ Supported interrupt.
⇒ 32 bit RISC

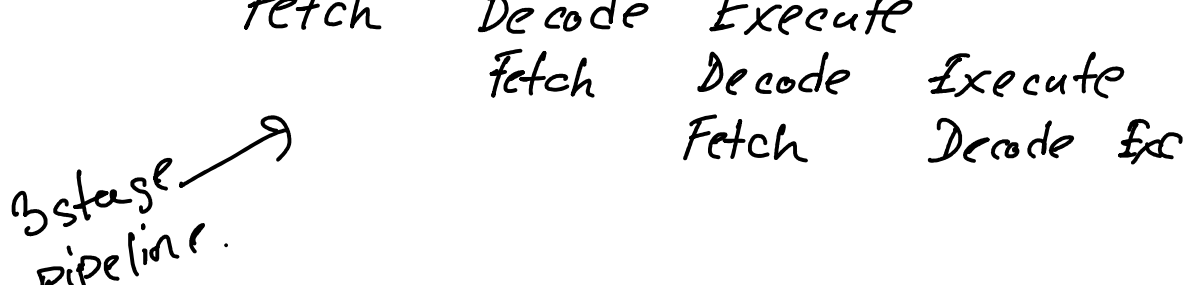
- supported interrupt.

⇒ 32 bit RISC

⇒ NVIC

- Nested Vectored Interrupt Controller
- manages & handles interrupts.
- provides a way to prioritize.
- Receives interrupt request & prioritize them according to priority level.
- Forwards interrupts according to ISR (interrupt service routine)
- Can handle 240 external interrupt & 16 internal u

Thumb is an instruction set. Improves code density & reduce memory. Uses a decoder to interpret 32 bit ARM instructions to 16 bit thumb instructions.



5 stage pipeline.

SIMD

- Single Instruction Multiple Data
- It's an unit that can perform SIMD instructions in a single cycle.

Some SIMD instructions:

(i) Add / Subtract : 8/16/32 bit

(ii) Multiplication : 8/16/32 bit

(iii) Multiply accumulate instruction.

(iv) Shift / Rotate

(v) Logical instruction

- MAC instruction
- performs both multiplication & addition in a single cycle.
- commonly used in DSP
- In a MAC operation the processor multiplies two values then adds the value of product to an accumulator register

the value of product to an accumulator register.

Accumulator = Accumulator + (Operand1 × Operand2)

↳ MAC reduces number of instructions.

SIMD maximizes register usage.
↳ Parallelizes operations.
↳ works with packed data.

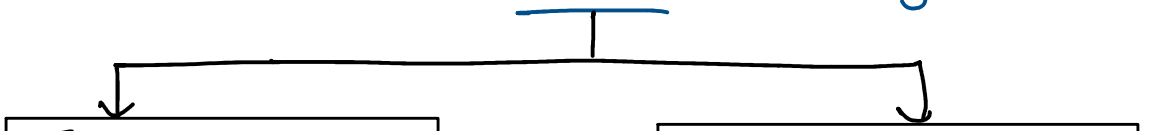
Cortex M4 has two operating states.

ARM
32 bit
word Aligned

Thumb
16 bit
halfword aligned

Operating state can be exchanged using the BX instruction.
↳ Branch & exchange.

Cortex M4 has two Operating modes.



Thread Mode
↳ executes application software

Handler Mode
↳ handles exception
↳ returns to thread

software
↳ Processor enters thread mode when it comes out of reset

↳ returns to thread mode when its done handling.

MSR & MRS instruction:

- MSR - Move to System Register
- MRS - Move from " "
- ↳ used to read from & write to system.

MRS <Rd>, <sysreg>
- <Rd> ↓ destination general purpose register
- <sysreg> ↗ name of system register being read

IPSR - Interrupt Program Status Register
↳ holds the exception number of currently executing ISR
↳ Interrupt Service Routine.

MSR <sysreg>, <Rn>
- <Rn> ↓ Source general purpose register

Unprivileged level:

→ limited access to MSR & MRS
→ No access to system timer, NVIC
→ Must use SVC instruction to make supervison call.
↓
supervisor call

62B4B42C-C324-42B...

Register Sets

- R0–R12:
  - General-Purpose Registers
  - 32-bit general-purpose registers for data operations or for addressing
  - R0-R7: Low registers, 16-bit Thumb instructions can only access to these
  - R8-R12: High registers, accessible by all 32-bit instructions
- Register R13
  - used as the Stack Pointer (SP)
  - two modes
  - SP_main: main stack pointer (MSP)
    - default SP
    - used by OS-kernel, exception handler, application code with privilege
  - SP_process: process stack pointer (PMSP)
    - usually in Thread mode
    - used by application code not running exception handler
  - PUSH and POP instruction used for access stack memory
  - full descending stack arrangement
  - SP decrements when new data is stored in stack
  - PUSH and POP are always word-aligned
  - Example: PUSH R0 ; Memory[R13]=R0
    POP R0 ; R0=memory[R13], R13=R13+4

Slide 3

Single Cycle MAC unit :
↳ multiply-accumulate

2) A digital circuit that performs multiplication & addition operations in a single clock cycle.

in a single clock cycle.

⇒ MAC unit take 2 input operands typically two n bit fixed point numbers. and performs the following computation

output = (input1 * input2) + accumulator

↓

register that holds the previous state or result of MAC op.

Outputting result & updating the accumulator takes 1 single cycle.

MAC is used in DSPs.

4D40B89D-9338-488...

Optimized MAC has very cool implementations damn.

- SMULBB: Multiplies two signed 8-bit integers and stores the 16-bit result with the bottom-bottom multiplication.
- SMULBT: Multiplies two signed 8-bit integers and stores the 16-bit result with the bottom-top multiplication.
- SMULTB: Multiplies two signed 8-bit integers and stores the 16-bit result with the top-bottom multiplication.
- SMULTT: Multiplies two signed 8-bit integers and stores the 16-bit result with the top-top multiplication.
- SMULABB: Multiplies two signed 8-bit integers, adds the result to the accumulator, and stores the 16-bit result with the bottom-bottom multiplication.
- SMULABT: Multiplies two signed 8-bit integers, adds the result to the accumulator, and stores the 16-bit result with the bottom-top multiplication.
- SMULATB: Multiplies two signed 8-bit integers, adds the result to the accumulator, and stores the 16-bit result with the top-bottom multiplication.
- SMULATT: Multiplies two signed 8-bit integers, adds the result to the accumulator, and stores the 16-bit result with the top-top multiplication.
- SMULALBB: Multiplies two signed 8-bit integers, adds the result to a 32-bit accumulator, and stores the 32-bit result with the bottom-bottom multiplication.
- SMULALBT: Multiplies two signed 8-bit integers, adds the result to a 32-bit accumulator, and stores the 32-bit result with the bottom-top multiplication.
- SMULALTB: Multiplies two signed 8-bit integers, adds the result to a 32-bit accumulator, and stores the 32-bit result with the top-bottom multiplication.
- SMULALTT: Multiplies two signed 8-bit integers, adds the result to a 32-bit accumulator, and stores the 32-bit result with the top-top multiplication.

C5A171B5-9954-4C9...

8A0C7ECE-9E79-488...

css

Copy code

```
; Multiply signed 8-bit integers x and y using SMULBB
; Result is stored in a 16-bit register

        SXTB    r0, r0      ; Sign-extend x from 8 to 32 bits
        SXTB    r1, r1      ; Sign-extend y from 8 to 32 bits
        SMULBB  r2, r0, r1  ; Multiply x and y with bottom-bottom multiplication
```

In this example, the SMULBB instruction is used to multiply two signed 8-bit integers that are stored in registers r0 and r1, respectively. The SXTB instruction is used to sign-extend the 8-bit values to 32 bits, which is required by the SMULBB instruction. The result of the multiplication is stored in a 16-bit register r2.

8-bit values to 32 bits, which is required by the SMULBB instruction. The result of the multiplication is stored in a 16-bit register r2.

Note that the SMULBB instruction performs the multiplication operation on the least significant bits of the input values, and the result is also stored in the least significant bits of the output register. Other instructions, such as SMULBT, SMULTB, and SMULTT, perform the multiplication operation on different combinations of the input bits, resulting in different output formats.

Table 1: Cortex-M4 single cycle MAC

Operation	Instruction	M3	M4
16 x 16 =32	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
16 x 16+32 =32	SMULABB, SMULABT, SMULATB, SMULATT	n/a	1
16 x 16+64 =64	SMULALBB, SMULALBT, SMULALTB, SMULALTT	n/a	1
32 x 32 =32	MUL	1	1
32 x 32 =64	SMULL,UMULL	5-7	1

n/a → not applicable

latency of SMULL is M3 is 5-7 clock cycles

latency - time delay between initiation & completion of operation.

Slide 4

ARM has 7 different operating modes: (7)

(1) User mode : In this process, processor runs in non privileged state. This mode is used for

runs in non privileged state. This mode is used for running most of the application of the system.

② FIQ Mode: In this mode processor is optimized for handling with low latency. FIQs are usually used for high priority tasks.

③ IRQ Mode: In this mode the processor is designed to handle normal interrupts with low latency. Used for handling asynchronous events.

- I/O operations
- timer events

④ Supervisor mode: Processor runs in privileged mode. Used for handling system level events & performing privileged operations.

⑤ Abort mode: Processor designed to handle memory related errors, page faults & bus errors.

⑥ Undefined mode: Processor enters an undefined state if

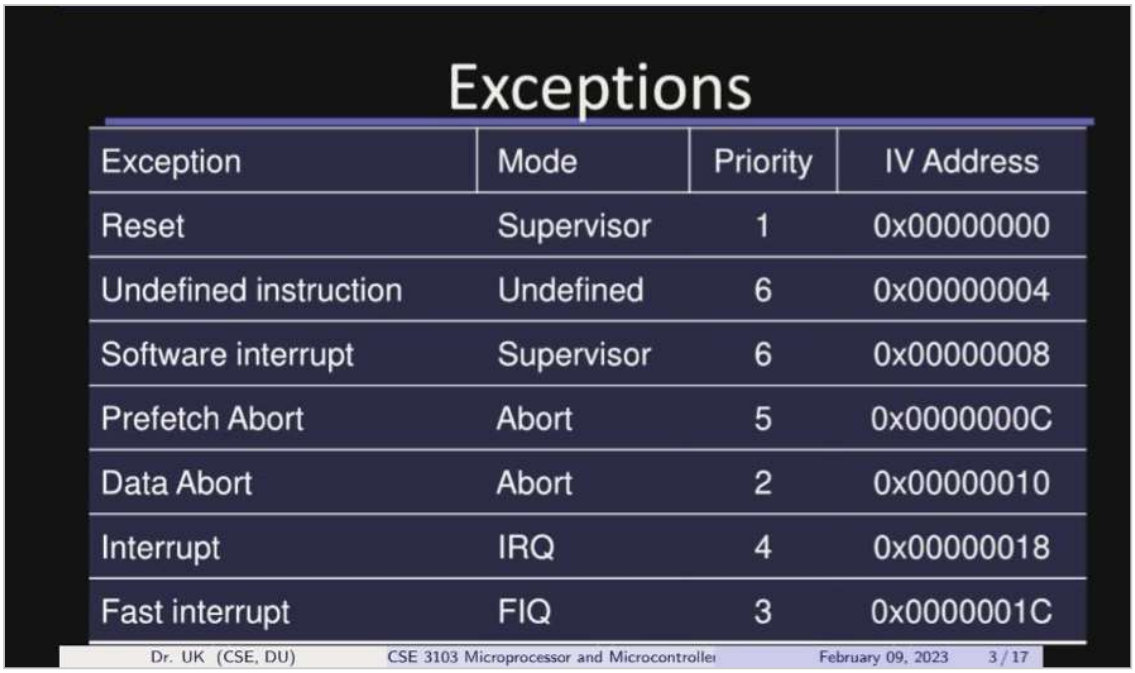
undefined state if encountered an invalid instruction. Used for debugging & testing purpose.

⑤ System Mode: Operating system mode, privileged mode

Exceptions

91A43D88-A25F-4FC...

2CA56741-3D1E-4B2...



1. Interrupts: External devices or software can interrupt the normal operation of the processor, and the processor temporarily suspends the current operation, saves the current state, and executes an interrupt service routine (ISR) to handle the interrupt.
2. Data aborts: The processor encounters a problem while accessing data, such as a page fault or a memory access violation, and switches to abort mode to execute an abort handler to handle the error.
3. Prefetch aborts: The processor encounters a problem while fetching instructions, such as an invalid instruction or a fetch error, and switches to abort mode to execute an abort handler to handle the error.
4. System calls: User-level software can request privileged operations, such as opening a file or allocating memory, and the processor switches to supervisor mode to execute a system call handler to handle the request.
5. Undefined instructions: The processor encounters an invalid instruction, and switches to undefined mode to execute an undefined instruction handler to handle the error.
6. Exceptions from coprocessors: Specialized hardware units called coprocessors can generate exceptions that are handled by the processor in a similar way to other types of exceptions.

Lecture 5

Basic features of ARMv7 :

① 32 bit RISC processor core

② 37 pieces of 32 bit integer registers.

③ Thumb instruction set

④ Pipelined (ARM7 : 3 stages)

(5) Debug interface

(6) Jazelle DBX: direct bytecode execution

(7) 7 modes of operation

i) usr ii) fiq iii) irq iv) svc

v) abt vi) sys vii) und

ARMv7 is Thumb Only profile.

Cortex M4 is a low power processor that features low gate count.

The Cortex M4 processor includes:

1) A Processor core

2) A NVIC → achieves low latency interrupt processing

3) An optional MPU - Memory Protection Unit.

4) An FPU - floating Point Unit.

5) Multiple High performance BUS interfaces.

6) Low cost debug solution

Processor incorporates 3 external BUS interfaces.

1) AHB — Advanced High-Performance Bus

2) APB — Advanced Peripheral Bus

3) FMI — External Memory Interface

FPU

⇒ 32 bit instructions

⇒ Combined MAC instruction for increased precision

⇒ 32 dedicated 32 bit single precision registers.

⇒ Decoupled three stage pipeline.

MPU

⇒ 8 memory regions

⇒ SRD (Sub Region Disable), efficient use of memory regions

NVIC

⇒ Achieves low latency interrupt processing

↳ amount of time takes for a system or device to respond to an input or request.

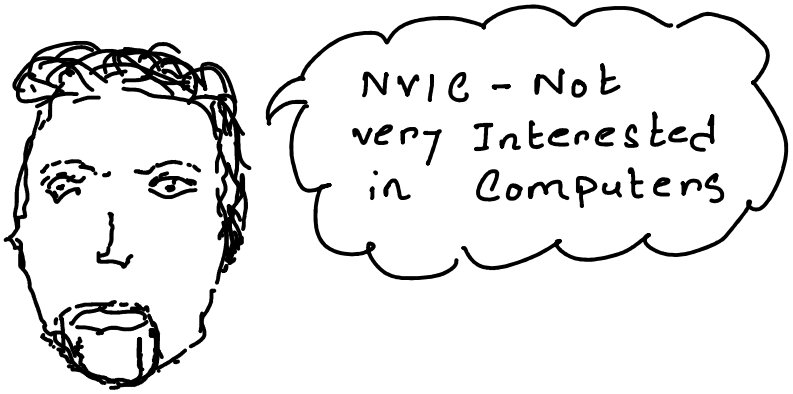
⇒ External interrupts configurable from 1 to 240.

⇒ Each external interrupt line corresponds to a specific hardware pin on the microcontroller that can be connected to an external event.

⇒ 1 to 240 means many configuration possible which is beneficial.

⇒ NVIC की ability is an important

→ NVIC का यह ability is an important feature.



→ Dynamic reprioritization of interrupts

→ Priority grouping

→ Support for tail chaining & late arrival of interrupts.

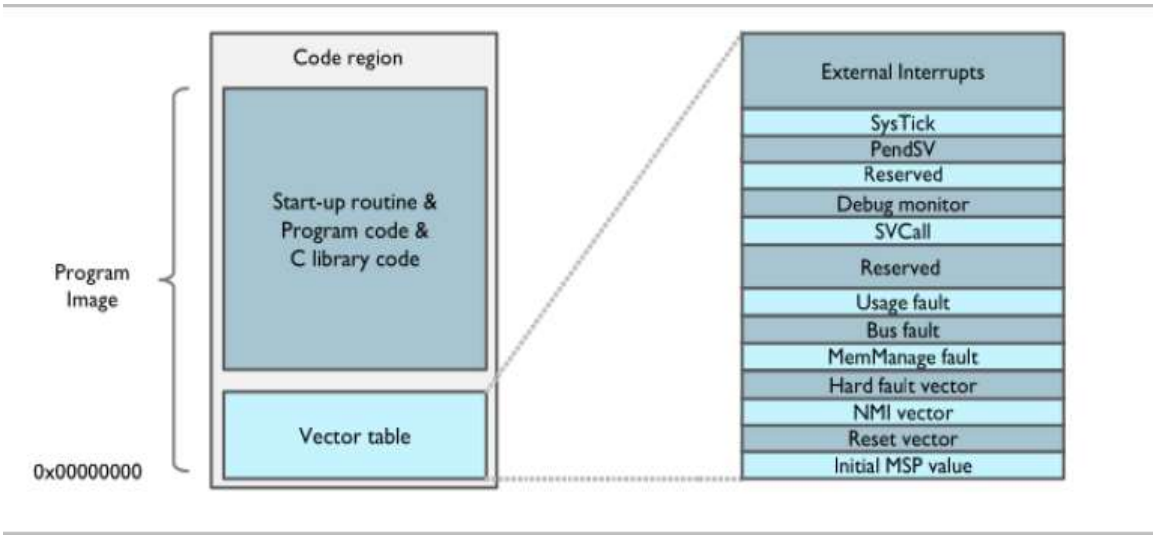
→ WIC - WaKup Interrupt Controller.

Lecture 6

ISA - Instruction Set Architecture

Vector table includes the starting addresses of exceptions (vectors) and the value of the main stack

point.

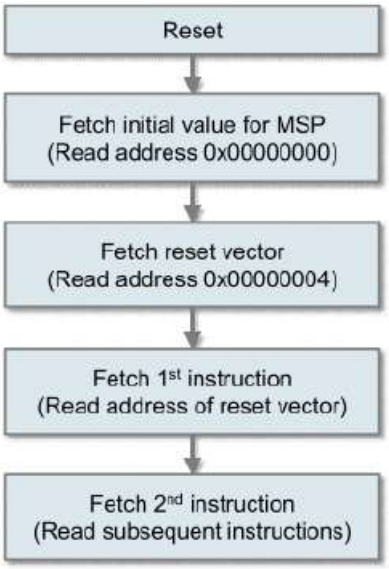


MSP- Main Stack Pointer

MSP register holds the current stack pointer value for the main stack, which is the stack used by the processor during exception handling and interrupt service routines.

The main stack pointer is initialized on Reset.

When an exception or interrupt occurs, the processor saves the current value of the program counter and other important registers onto the stack pointed to the MSP.



Endianness:

Supports both little endian & big endian.

In a little-endian format, the least significant byte (LSB) of a multi-byte data value is stored at the lowest memory address, while the most significant byte (MSB) is stored at the highest memory address. For example, a 32-bit data value "0×12345678" would be stored as "0×78" at the lowest memory address, followed by "0×56", "0×34", and "0×12" at the highest memory address.

In a big-endian format, the MSB of a multi-byte data value is stored at the lowest memory address, while the LSB is stored at the highest memory address. For example, the same 32-bit data value "0×12345678" would be stored as "0×12" at the lowest memory address, followed by "0×34", "0×56", and "0×78" at the highest memory address.

- Endian refers to the order of bytes stored in memory
  - Little endian: lowest byte of a word-size data is stored in bit 0 to bit 7
  - Big endian: lowest byte of a word-size data is stored in bit 24 to bit 31

Main Features of Cortex M4 Instruction Set

- All instructions are 32 bits long.
- Supports 32-bit Thumb-2 instructions
- Possible to handle all processing requirements in one operation state (Thumb state)
- No need to separate ARM code and Thumb code source files, which makes the development and maintenance of software easier
- Most instructions execute in a single cycle.
- Every instruction can be conditionally executed.
- A load/store architecture
  - Data processing instructions act only on registers
    - Three operand format
    - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes.
    - 32 bit and 8 bit data types
    - Flexible multiple register load and store instructions

operand: can be an ARM Cortex M4 register, a constant, or another instruction-specific parameter.

- For data processing instruction, first operand is the destination (destination register) of the operation
- For memory read instruction (except multiple load instruction), first operand is the register which data is loaded into
- For memory write instruction (except multiple store instruction), first operand is the register that holds the data to be written to the memory

Conditional execution :

Addition based on zero flag:

```
CMP  R3, #0
ADDNE  R0, R1, R2
ADDEQ
```

Compares R3 with 0, then adds R1 & R2 if R3 is not equal to 0. and adds R1 & R3 if R3 is equal to 0.

Conditional Branch based on negative flag:

```
CMP   R0, #0
BPL   label1
 B    label2

label1:  ; code to execute if R0 >=0

label2:  ; code to execute if R0 <0
```

- SUBCS - Subtraction with carry set
- SUBCC - Subtraction with carry clear

Conditional subtraction:

```
CMP   R1, R2
SUBCS R0, R1, R2
SUBCC R0, R1, R3
```

SUBCS stores in R0 if carry flag is not set.

SUBCC stores the result in R0 if carry flag is set

Compare three value & store the smallest one:

```
; Compares R0, R1, and R2 and
stores the smallest value in R3

CMP  R0, R1 ; Compare R0 & R1
MOVLT R3, R0 ; IF R0<R1, move
                R0 to R3
MOVGE R3, R1 ; IF R0>=R1, move R1 to R3
CMP  R3, R2
MOVGT R3, R2 ; IF R3 > R2 move
                R2 to R3.
```

CMP instruction compares two register values by performing a subtraction between them without updating the result in the register. CMP updates the condition flags. N, Z, C, V.

assembly

Copy code

```
; Let's assume R0 and R1 have been assigned some values
CMP   R0, R1     ; Compare R0 and R1 (R0 - R1), update condition flags
BEQ   Equal      ; If R0 == R1 (zero flag set), branch to Equal label
B    NotEqual    ; If R0 != R1, branch to NotEqual label

Equal:
    ; Perform some action when R0 and R1 are equal
    ; ...
    B    Exit      ; Branch to Exit label after performing the action

NotEqual:
    ; Perform some action when R0 and R1 are not equal
    ; ...

Exit:
    ; Continue the rest of the program
```

Yes, you can check greater than or less than using the `CMP` instruction by examining the appropriate condition flags after the comparison. Here's an example:

assembly Copy code

```
; Let's assume R0 and R1 have been assigned some values
CMP   R0, R1      ; Compare R0 and R1 (R0 - R1), update condition flags
BGT   Greater     ; If R0 > R1 (negative flag == overflow flag), branch to Gr
BLT   Less        ; If R0 < R1 (negative flag != overflow flag), branch to Le
B    Equal        ; If R0 == R1, branch to Equal label (equal case)

Greater:
    ; Perform some action when R0 > R1
    ; ...
    B    Exit       ; Branch to Exit label after performing the action

Less:
    ; Perform some action when R0 < R1
    ; ...
    B    Exit       ; Branch to Exit label after performing the action

Equal:
    ; Perform some action when R0 and R1 are equal
    ; ...

Exit:
    ; Continue the rest of the program
```

ARM Condition Codes

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Addressing modes

CPU determines the memory address by adding a positive or negative offset to the value in a base register.
The way in which CPU combines these two parts is called Addressing mode.

ARM instruction set has (3) addressing modes.

ARM instruction set has (3) addressing modes.

It says we have 3 addressing modes:

1. Register Addressing
2. Immediate Addressing
3. Memory Addressing
   - Register offset
   - PC relative addressing
   - Indexed addressing
     - Load & store multiple register

Immediate Addressing:

Way to directly use constant values as operands in an instruction, without the need for loading them from memory or registers.

Makes execution faster but limits possible values.

Adding a constant value to register:

```
ADD R0, R1, #5
```

Performing a bitwise AND operation:

```
AND R2, R3, #0xFF
```

$(F)_{16} = (1111)_2$

$(0xFF)_{16} = (1111\ 1111)_2$

→ Performs bitwise operation between R3 & 0xFF and stores to R2.

Comparing a value:

```
CMP R4, #10
```

Register Addressing

→ unsigned integer

An "offset" is a value that represents the difference or distance between two memory addresses or locations. In terms of addressing modes, an offset is often used to calculate the effective memory address of accessing data.

```
LDR R0, [R1, R2]
```

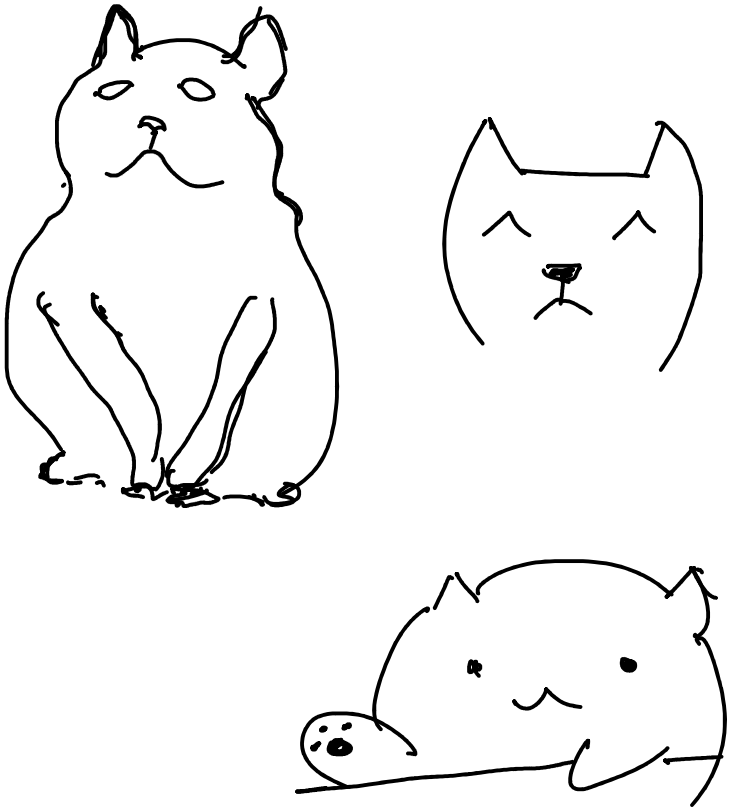
→ contains the offset value.

[illegible] contains the offset value.

contains the base address

Load the value from memory address (R1 + R2) to R0.

There are various types of register addressing:



· Types of register addressing:

Register direct addressing:

Instruction directly uses a register as an operand.

Instruction directly uses a register as an operand.

```
ADD R0, R1, R2
```

Register Indirect Addressing:

Address of the operand is stored in a register. The instruction refers to the register to access the data in memory.

```
LDR R0, [R1] ; Load the value
```

from the memory address stored in R1 into R0.

Register Offset addressing:

A base register & an offset register is used to calculate the effective memory address. Data is accessed from resulting memory access.

```
LDR R0, [R1, R2] ; Load value
    from the memory address
    R1+R2 into R0.
```

Register Pre Indexed Addressing:

Base register & offset register.
Base register is updated with the resulting memory address before the data transfer

LDR R0, [R1, #4]!

LDR R0, [R1, R2]!
Load the value from memory address (R1+R2) into R0, and update with R1 + R2.

Register Post indexed addressing:

Same as pre indexed. updated after the data transfer.

STR R2, [R3], R4;
Store the value in R2 at the memory address in R3 and update R3 with (R3 + R4) after memory access
Simply,
Store the value in R2 at the memory address specified in R3
After the value is stored add the value in R4 to the value in R3
Update R3 with the result of the addition from step 2.

Binary Encodings

C.4 from Sarah Harris

3 main instruction format

1. Data Processing
2. Memory
3. Branch

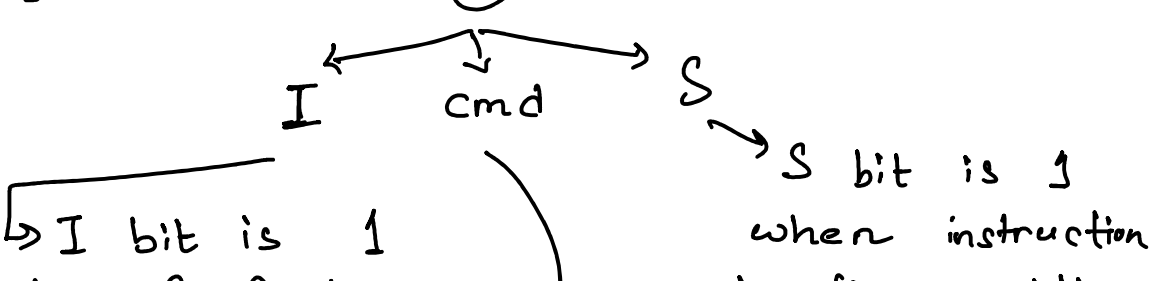
Data Processing Instructions

The 32 bit instruction set has six fields :

cond, op, funct, Rn, Rd, Src 2

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src 2
4	2	6	4	4	12

funct has 3 subfields.



↳ I bit is 1 when Src2 is an immediate

when instruction sets the condition flags.

indicates the specific data processing instruction. Suppose cmd is $4_{(0100)_2}$ for ADD.

Three variations of Src2 encoding allow the second source operand to be (1) an immediate

(2) a register optionally shifted by a constant (shamt5)

(3) A register (Rm) shifted by another register (Rs)

(sh) → shift

Table 6.8 *sh* field encodings

Instruction	sh	Operation
LSL	00_2	Logical shift left
LSR	01_2	Logical shift right
ASR	10_2	Arithmetic shift right
ROR	11_2	Rotate right

Three variations of (Src2)

1. Immediate value as second source operand

```
ADD R0, R1, #10 ; R0 = R1+10
```

#10 → Second source operand

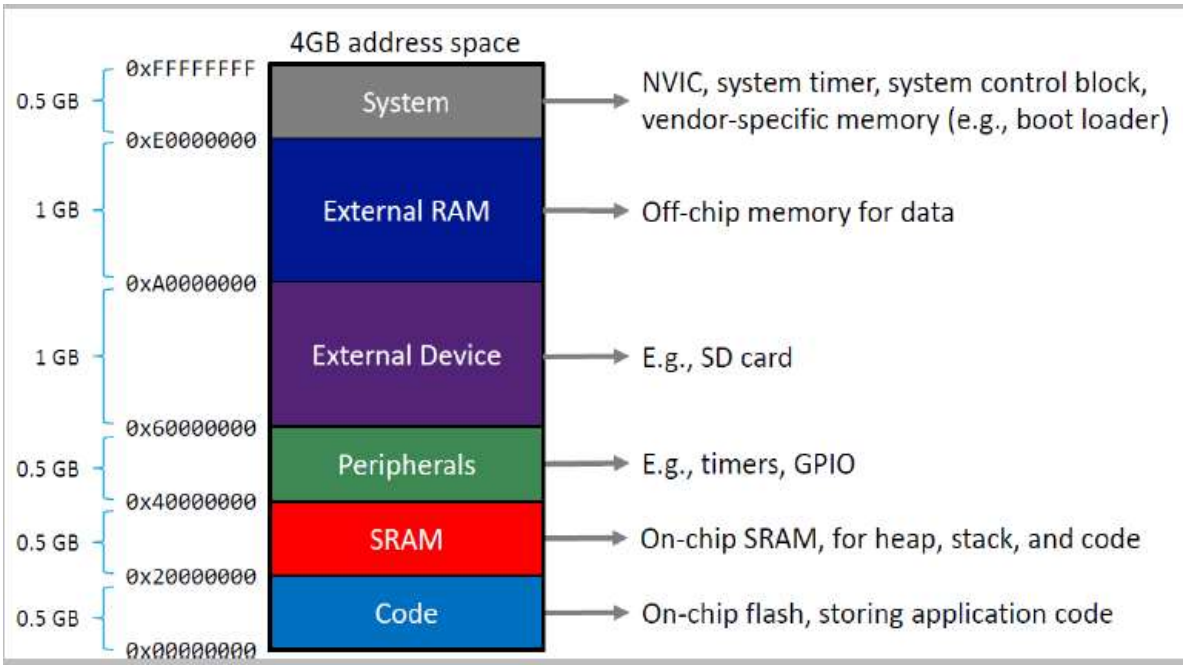
2.

So, this is maybe page 339 from Book 2. Will continue from there. And lecture 6 is done except

And lecture 6 is done except the Binary Encoding. Don't miss that.

Lecture 7 later

Lecture 8



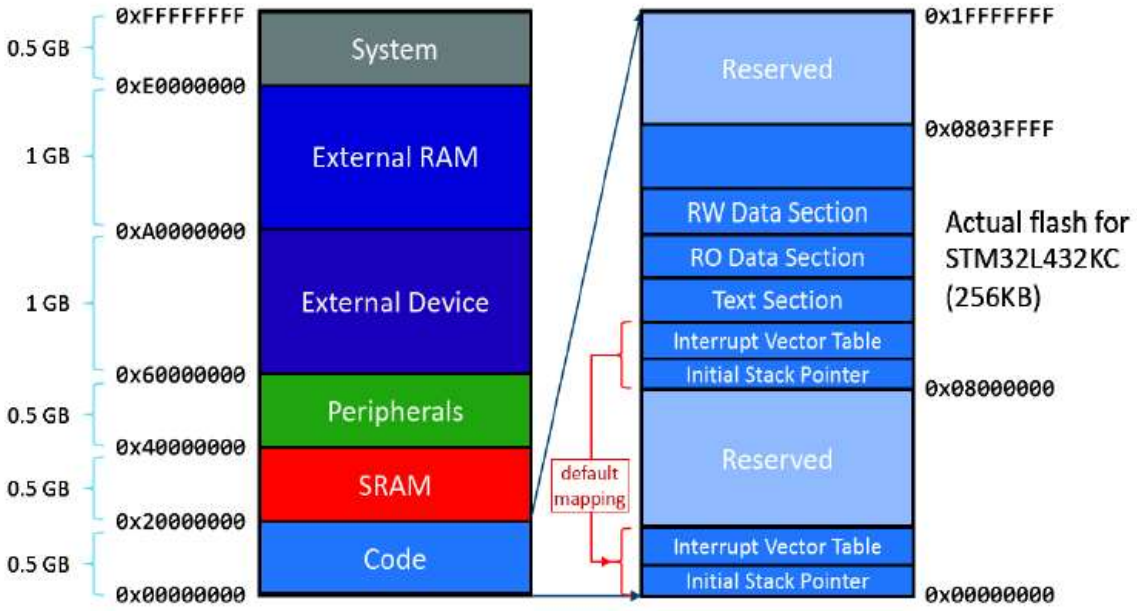


Figure: Processor Memory Model

Bus matrix ensures communication between various internal & external elements.

SCS - System Control Space

- Masters - CPU, DMA controller
- Slaves - Memory, peripherals

→ Bus matrix ensures smooth data transfer between them.

SCS has system level function controlling registers. like: NVIC, Systick timer.

Priority is given to the processor.

Memory Regions :

Memory Map → Region

Code → Instruction fetches are performed over the ICode bus.
Data instruction processes are performed over the DCode bus.

SRAM → Instruction fetch & Data accesses were performed over the System Bus.

SRAM bit band → Alias region.

Peripheral → System bus

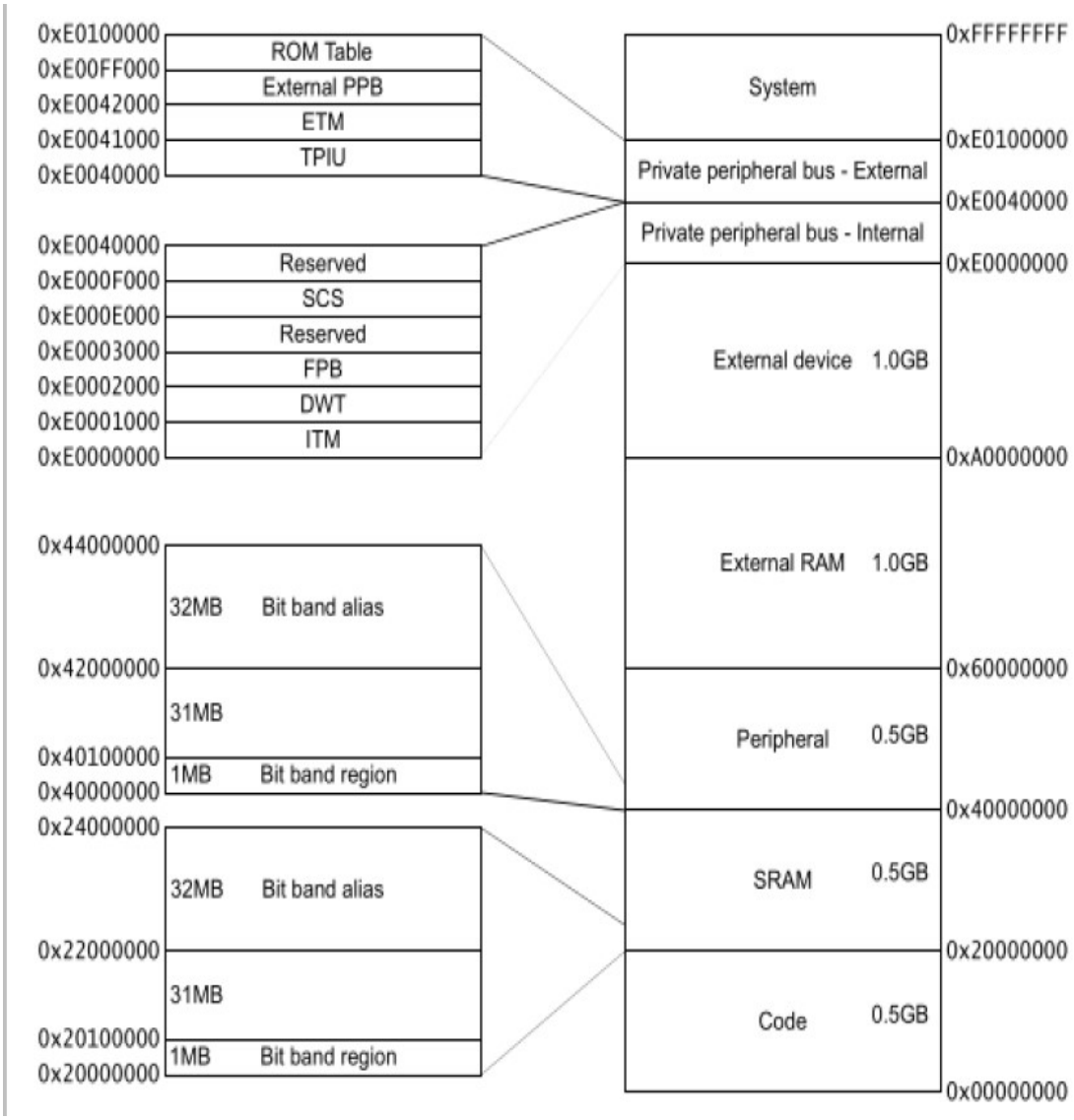
Peripheral bit band → Alias region

External RAM - System Bus

" Device - System "

A memory region is a contiguous block of memory address

System Address Map:



Cortex-M4 system address map:

1. Memory regions: The address map is divided into several regions, including Code (Flash), SRAM, Peripheral, External Device, External RAM, and System Control Space (SCS) regions.
2. Code (Flash) region: Located at address range 0x0000_0000 to 0x1FFF_FFFF. This region is used to store program code and read-only data. The Cortex-M4 fetches instructions from this region during execution.
3. SRAM region: Located at address range 0x2000_0000 to 0x3FFF_FFFF. This region provides general-purpose memory for storing variables and temporary data. It is also used for stack operations.
4. Peripheral region: Located at address range 0x4000_0000 to 0x5FFF_FFFF. This region contains memory-mapped registers for controlling and accessing peripherals like GPIO, UART, SPI, and timers.
5. External Device and External RAM regions: Located at address ranges 0x6000_0000 to 0x9FFF_FFFF (External Device) and 0xA000_0000 to 0xDFFF_FFFF (External RAM). These regions are used to interface with external devices and memory, such as external flash, EEPROM, or DRAM.
6. System Control Space (SCS) region: Located at address range 0xE000_0000 to 0xEFFF_FFFF. This region contains registers for system-level control and configuration, including NVIC, System Control Block (SCB), and SysTick Timer.
7. Private Peripheral Bus (PPB): Located within the SCS region, specifically at address range 0xE000_0000 to 0xE00F_FFFF. This region contains private peripherals such as the NVIC, SCB, MPU, and debug components.
8. Memory Protection Unit (MPU): The Cortex-M4 supports an MPU to control access to different memory regions, providing fine-grained control over memory permissions and improving system security.

PPB – Private Peripheral Bus

Provides access to internal & external processor resources.

Internal PPB provides Access to:

① ITM – Instrumentation Trace Macrocell.

② DWT – Data Watchpoint & Trace
③ FPB – Flashpath & Breakpoint.
④ SCS
⑤ MPU
⑥ NVIC

External PPB provides to:

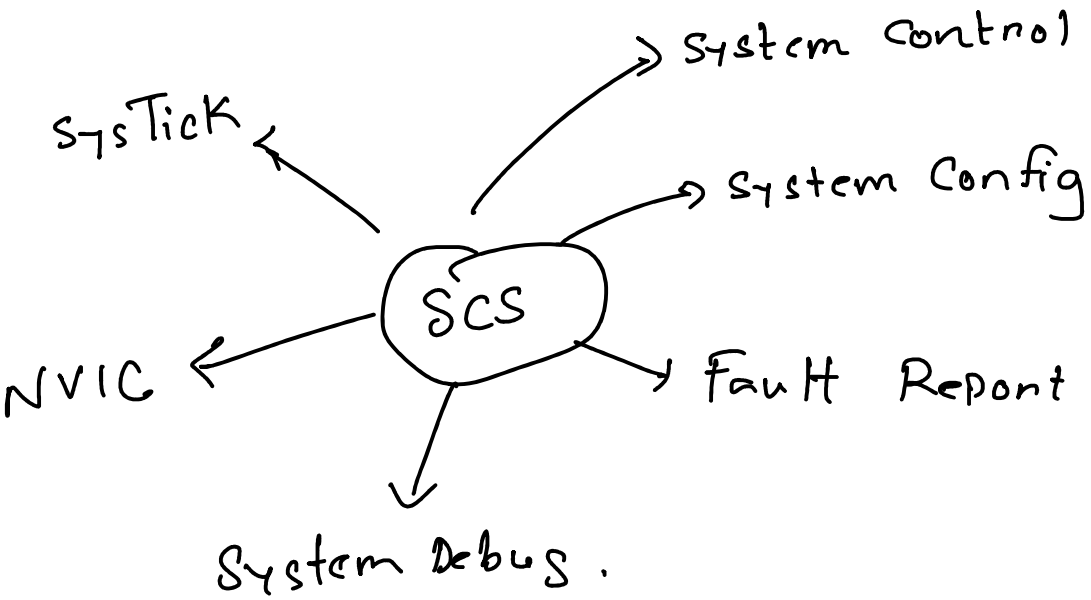
① TPIU ② ETM ③ ROM Table.

In the ARM Cortex-M4 memory map, the System region, also known as System Control Space (SCS), is located at the address range 0xE000_0000 to 0xEFFF_FFFF. This region contains registers and components related to system-level control, configuration, and debugging. Here are some key parts of the System region:

1. Nested Vectored Interrupt Controller (NVIC): The NVIC is a key component of the Cortex-M4's interrupt handling system. It handles prioritization, masking, and vectoring of interrupts from peripherals and other sources. The NVIC registers are located at the base address 0xE000_E000.
2. System Control Block (SCB): The SCB is responsible for controlling various aspects of the processor's operation, such as system resets, fault handling, and CPUID identification. The SCB registers are located at the base address 0xE000_ED00.
3. SysTick Timer: The SysTick timer is a 24-bit down-counter that can be used to generate periodic interrupts for tasks like timekeeping, scheduling, or performance measurement. Its registers are located at the base address 0xE000_E010.
4. Memory Protection Unit (MPU): The MPU is an optional component that provides fine-grained control over memory access permissions for different regions, improving system security. The MPU registers are located at the base address 0xE000_ED90.
5. Debug components: The System region also contains registers and components for debugging purposes, such as the Debug Halting Control and Status Register (DHCSR) and the Debug Exception and Monitor Control Register (DEMCR). These registers are located at the base addresses 0xE000_EDF0 and 0xE000_EDFC, respectively.
6. Private Peripheral Bus (PPB): The PPB is a part of the SCS region, located at the address range 0xE000_0000 to 0xE00F_FFFF. This area contains private peripherals like the NVIC, SCB, MPU, and debug components, which are crucial for system-level control and management.

SCS is a memory-mapped 4KB address space that provides 32 bit registers for configuration.

SCS registers divide into the following group



Bit banding

Bit band is a feature that allows atomic read-modify-write operations on individual bits in specific memory regions.

A bit band region is a contiguous area of memory where the processor supports bit banding.

In cortex M4 there are two bit band regions:

(1) SRAM bit band region - The region is located in the SRAM memory space.

(2) Peripheral bit band region: covers the peripheral memory space.

Calculating bit band alias address:

Alias address = Bit-band base + (byte_offset * 32) + (bit_number * 4)

byte_offset: number of bytes between the base address of the original memory region & the target byte.

bit_number: position of the target bit within the target byte.

Then an operation will be conducted on the alias address, a read write operation.

Alias address is a unique 32bit memory location representing a single bit on the memory region.

First, you need to know the following formula to calculate each bit (from bit-band region) alias address. This formula is adapted from Cortex-M3 technical reference manual:

- bit_word_offset = (byte_offset x 32) + (bit_number × 4)
- bit_word_addr = bit_band_base + bit_word_offset

Where:

- bit_word_offset is the position of the target bit in the bit-band memory region.
- bit_word_addr is the address of the word in the alias memory region that maps to the targeted bit.
- bit_band_base is the starting address of the alias region.
- byte_offset is the number of the byte in the bit-band region that contains the targeted bit.
- bit_number is the bit position (0-7) of the targeted bit.

Example

- Original start address of the stored variable: 0x20000004
- Base address of that bit band region (in SRAM): 0x20000000
- Base address of that bit band alias region (in SRAM): 0x22000000
- Total offset of stored variable from bit band region: 0x4
- Alias formula: (total offset * 0x20)+(number of bit you want to manipulate *4)= (0x4 * 0x20)+(3*4)=0x14
- Bit band alias memory location: 0x22000000+0x14=0x22000014
- So use 0x22000014 to manipulate the 3rd bit of memory location 0x20000004

Lecture 5

Interrupt handling sequence:

Step by step:

① Interrupt Request Generation (IRQ)
A peripheral or external

(1) Interrupt request (IRQ):

A peripheral or external resource generates a interrupt request to the processor.

(2) Interrupt detection:

The processor detects the IRQ and check if its enabled in the NVIC, if it is then proceeds to the next option or otherwise does the normal execution

(3) Interrupt prioritization:

NVIC checks the priority of the incoming IRQ against the current ongoing task. If IRQ has higher priority then processor goes to handle the interrupt.

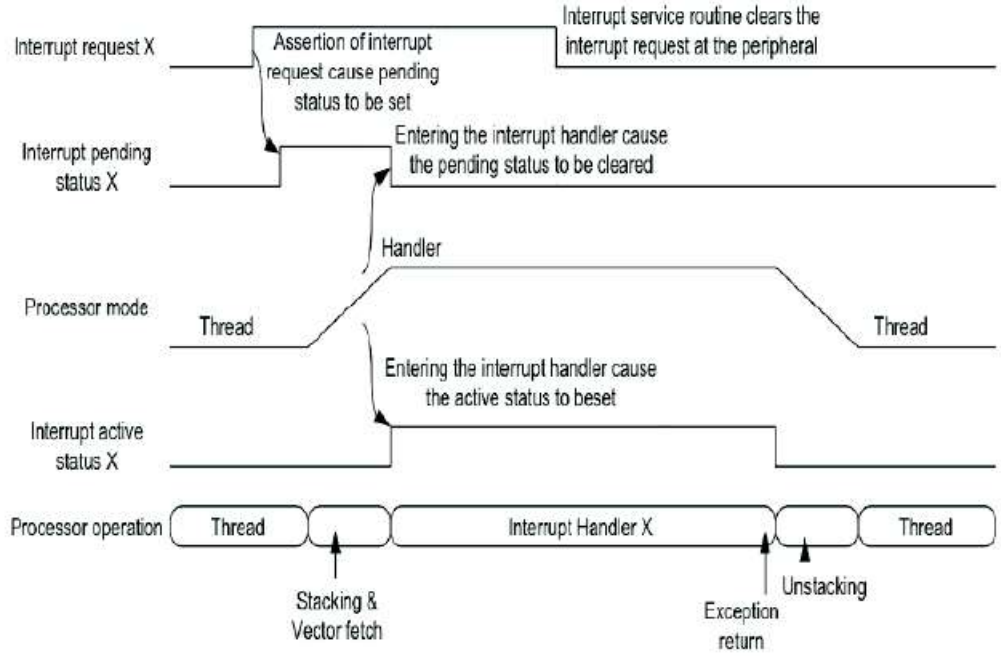
(4) Pre emption:

If IRQ has higher priority than current task then processor saves the current work progress and so on and then proceeds to handle interrupt.

(5) Vector table lookup:

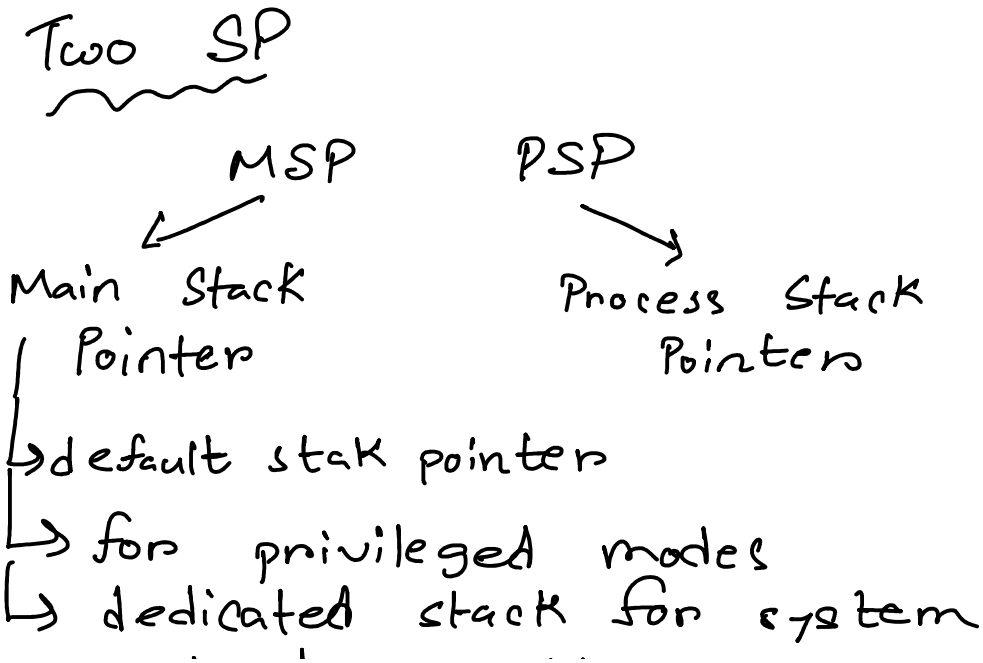
Processor performs a vector table lookup to find the ISR associated with the IRQ.

⑥ ISR execution: Interrupt Service Routine execution.



Handwired Exception processing sequence:

1. Finish current instruction (except for lengthy instructions): When an exception is detected, the processor completes the execution of the current instruction before entering the exception handling sequence. If the current instruction is a lengthy operation like a multiple-load or multiple-store, the processor might cancel the operation to handle the exception more quickly.
2. Push context (8 32-bit words) onto current stack (MSP or PSP): The processor saves the current execution context on the active stack (Main Stack Pointer or Process Stack Pointer). This context includes the following registers: xPSR, Return address, LR (R14), R12, R3, R2, R1, and R0. Saving the context allows the processor to restore the state after the exception handler has completed.
3. Switch to handler/privileged mode, use MSP: The processor switches to handler mode and starts using the Main Stack Pointer (MSP) for stack operations. Handler mode is a privileged mode that provides access to system resources and prevents unprivileged code from interfering with the exception handling process.
4. Load PC with address of exception handler: The processor looks up the address of the exception handler from the vector table and loads it into the Program Counter (PC) register. This directs the processor to start executing the exception handler code.
5. Load LR with EXC_RETURN code: The Link Register (LR) is loaded with an EXC_RETURN code, which is a special value that tells the processor how to return from the exception handler. This code specifies which stack pointer to use (MSP or PSP) and whether to return to privileged or unprivileged mode.
6. Load IPSR with exception number: The exception number is loaded into the Interrupt Program Status Register (IPSR), which is part of the xPSR register. This value helps the exception handler identify the source of the exception.
7. Start executing code of exception handler: With the setup complete, the processor begins executing the exception handler's code. Once the handler has completed, the processor restores the saved context and resumes normal execution from the point where the exception occurred.



↳ dedicated stack for system critical operations

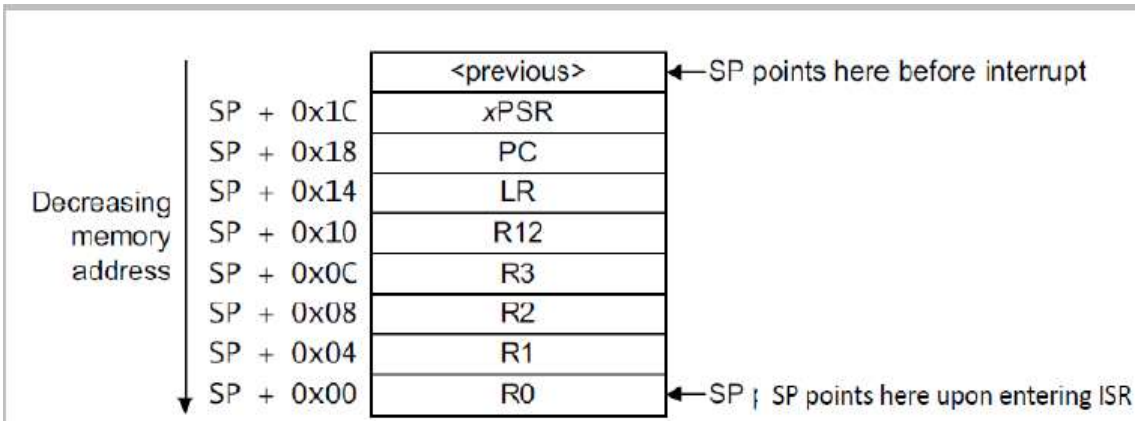
PSP

- ↳ for unprivileged mode
- ↳ u multitasking environments
- ↳ Task isolation

So basically

MSP - System level operation
PSP - Application u u

Stack grows toward small addresses.



Thread

Privileged or unprivileged

MSP or PSP

Handler

Handler

Privileged — MSP

IPSR- Interrupt Program Status Register.

- Holds currently active exception number
- part of APSR
  - part of xPSR
- Does nested exception handling.
- Context switch & exception return.
- Read only register

Once the IPSR is updated with the exception number, the processor proceeds to load the Program Counter (PC) with the address of the exception handler and starts executing the handler code. When the exception handler finishes its execution, the processor restores the context and returns to the normal program execution.

Vector table:

An array of addresses that points to the respective exception handlers. When an exception occurs, the processor looks up the corresponding handler address in the vector table and jumps to that address to

handler address in the vector table and jumps to that address to execute the exception handler.

how "Load PC with address of Exception Handler" works, describe with an example, draw a diagram if necessary and so on

When an exception occurs, the processor needs to load the address of the corresponding exception handler into the Program Counter (PC) so that it can start executing the handler. Here's how it works:

1. The processor detects an exception, such as an interrupt or fault, and determines its exception number.
2. The processor multiplies the exception number by 4, as each entry in the Vector Table is 4 bytes (32 bits).
3. The processor adds the result to the base address of the Vector Table (by default, 0x00000000, but it can be changed using the VTOR).
4. The calculated address points to the entry in the Vector Table for the corresponding exception handler. The processor reads the 32-bit value at that address, which is the starting address of the exception handler.
5. The processor loads the exception handler's address into the PC, effectively causing a jump to that address and starting the execution of the handler.

Example:

Let's say we have an external interrupt request with exception number 16. The default Vector Table base address is 0×00000000.

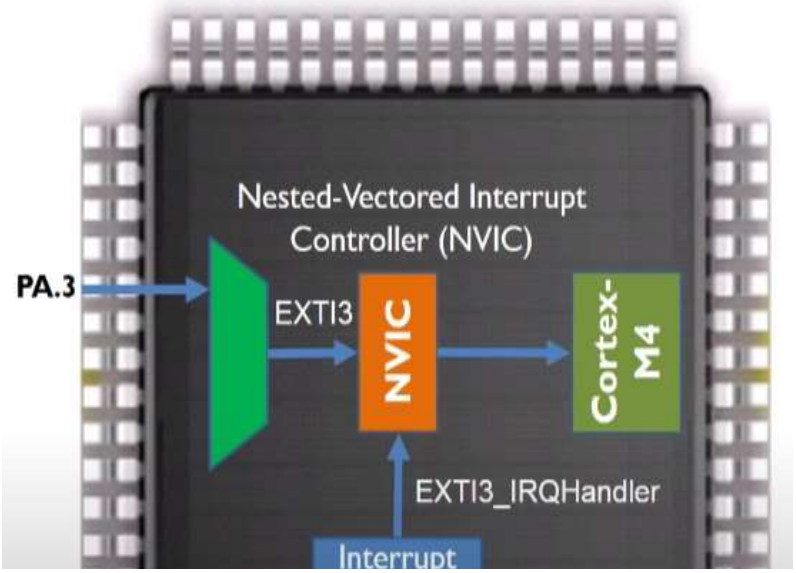
1. Exception number: 16
2. Multiply by 4: 16 * 4 = 64
3. Add to base address: 0×00000000 + 64 = 0×00000040
4. The processor reads the address of the exception handler at 0×00000040 in the Vector Table.
5. The processor loads the read address into the PC and jumps to the exception handler.

In summary, the processor calculates the address of the corresponding exception handler using the exception number, looks up the handler address in the Vector Table, and then loads it into the PC to start executing the handler.

0x000000000 + 64 = 0x00000040
In here decimal 64 is equal to
Hexa 40. okay?

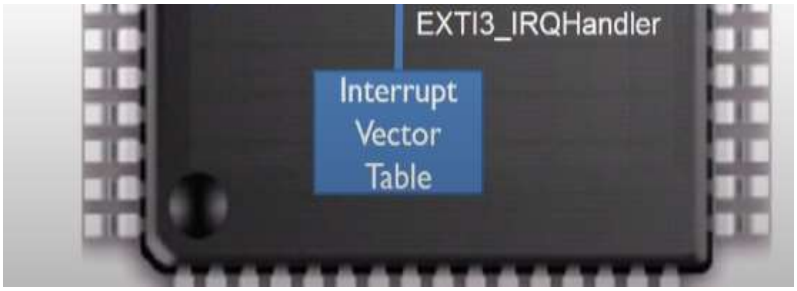
Youtube

Interrupts



NVIC prioritizes & handles all interrupts.

ISR or Interrupt handler



type এর কোনো
NVIC Interrupt
এর ভেক্টর টেবিল
নেই কারণ।

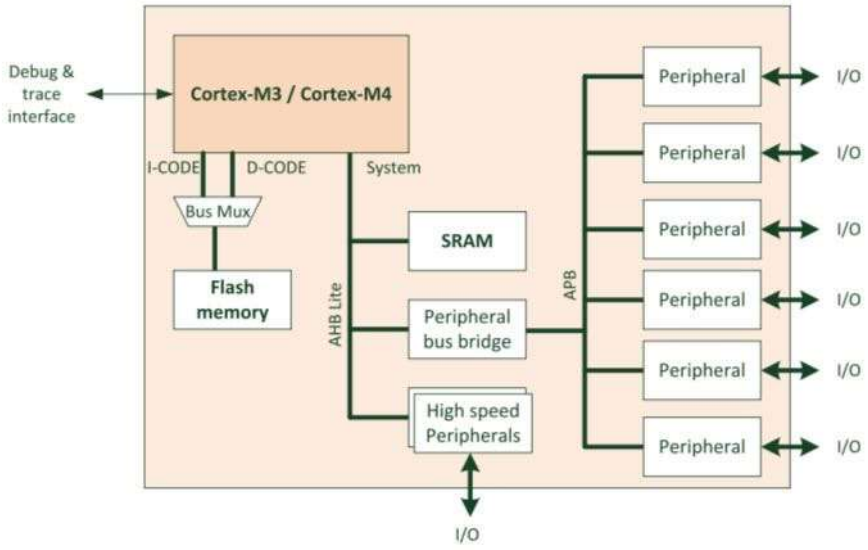
The entry points of all interrupt service routine are stored in Interrupt Vecton Table.

- holds an array of memory addresses
- each entry 4 bytes long

When interrupt x is triggered, jump to the ISR for interrupt x
$1 \leq x \leq 255$

How NVIC controller use interrupt numbers to look up the interrupt-vector table.

Main stack typically located in SRAM



JRE 6.3

mple Cortex-M3 or Cortex-M4 processor based system

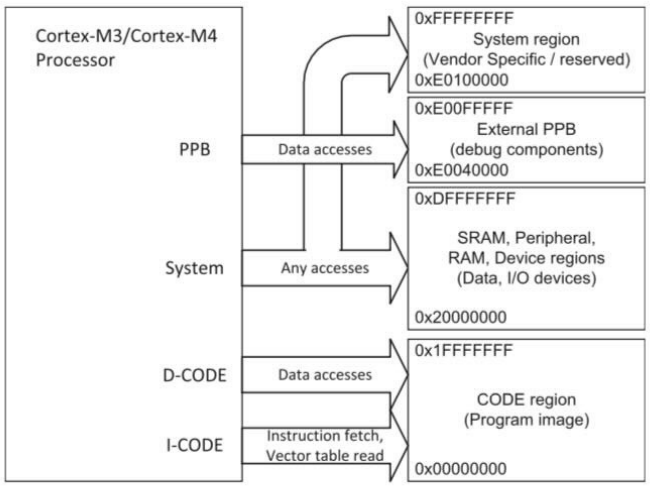


FIGURE 6.2

Multiple bus interface for different memory regions

Random abbreviations

Sunday, June 4, 2023 1:08 PM

General-Purpose Registers:

R0, R1, R2, ..., R12: General-purpose registers used for data storage and manipulation.
SP: Stack Pointer register, used to manage the stack.
LR: Link Register, used for storing the return address of function calls.
PC: Program Counter, holds the address of the next instruction to be executed.

Special-Purpose Registers:

APSR: Application Program Status Register, holds various status flags such as condition code flags.
IPSR: Interrupt Program Status Register, indicates the active interrupt or exception number.
MSP: Main Stack Pointer, used for the main stack in the Cortex-M3 and Cortex-M4.
PSP: Process Stack Pointer, used for the process stack in the Cortex-M3 and Cortex-M4.

System Control Space Registers (SCS):

VTOR: Vector Table Offset Register, used to relocate the vector table.
AIRCR: Application Interrupt and Reset Control Register, controls system reset and interrupt behavior.
SCR: System Control Register, controls system behavior and mode configuration.
CCR: Configuration Control Register, controls various configuration options.
SHPR1, SHPR2, SHPR3: System Handler Priority Registers, hold priority levels for various exceptions.
NVIC_ISERx: Interrupt Set Enable Registers, used to enable specific interrupts.

Memory Protection Unit (MPU) Registers (Cortex-M4 only):

MPU_TYPE: MPU Type Register, provides information about the MPU.
MPU_CTRL: MPU Control Register, enables/disables the MPU and controls its behavior.
MPU_RNR: MPU Region Number Register, selects the region for configuration.
MPU_RBAR: MPU Region Base Address Register, sets the base address for a region.
MPU_RASR: MPU Region Attribute and Size Register, configures attributes and size for a region.

NVIC Registers:

NVIC_ISERx: Interrupt Set Enable Registers, enable specific interrupts.
NVIC_ICERx: Interrupt Clear Enable Registers, disable specific interrupts.

NVIC_ISPRx: Interrupt Set Pending Registers, set specific interrupts as pending.
NVIC_ICPRx: Interrupt Clear Pending Registers, clear pending status of specific interrupts.
NVIC_IPRx: Interrupt Priority Registers, set priority levels for specific interrupts.
NVIC_STIR: Software Trigger Interrupt Register, triggers a specific interrupt by software.

SCB Registers:

SCB_CPUID: CPU ID Base Register, provides information about the processor.
SCB_ICSR: Interrupt Control and State Register, controls interrupt behavior and provides status information.
SCB_VTOR: Vector Table Offset Register, sets the base address of the vector table.
SCB_AIRCR: Application Interrupt and Reset Control Register, controls system reset and interrupt behavior.
SCB_SCR: System Control Register, controls system behavior and configuration.
SCB_CCR: Configuration Control Register, configures various system options.
SCB_SHPRx: System Handler Priority Registers, set priority levels for system exceptions.

MPU Registers (Memory Protection Unit, Cortex-M4 only):

MPU_TYPE: MPU Type Register, provides information about the MPU.
MPU_CTRL: MPU Control Register, enables/disables the MPU and controls its behavior.
MPU_RNR: MPU Region Number Register, selects the region for configuration.
MPU_RBAR: MPU Region Base Address Register, sets the base address for a region.
MPU_RASR: MPU Region Attribute and Size Register, configures attributes and size for a region.

Flash Memory Control Registers:

FLASH_ACR: Flash Access Control Register, controls the behavior and configuration of the flash memory.SysTick Timer Registers:

SysTick_CTRL: SysTick Control and Status Register, controls the behavior of the SysTick timer.
SysTick_LOAD: SysTick Reload Value Register, sets the initial count value for the SysTick timer.
SysTick_VAL: SysTick Current Value Register, holds the current count value of the SysTick timer.

PendSV Registers:

PSP: Process Stack Pointer, used for the process stack in the PendSV exception.
MSP: Main Stack Pointer, used for the main stack in the PendSV exception.

FPU (Floating-Point Unit) Registers (Cortex-M4 with FPU):

CPACR: Coprocessor Access Control Register, enables/disables access to the FPU registers.

FPCCR: Floating-Point Context Control Register, controls the FPU context and rounding modes.
FPCAR: Floating-Point Context Address Register, holds the address of the FPU context.
FPDSCR: Floating-Point Default Status and Control Register, controls default FPU operation behavior.

Power Control Registers:

PWR_CR: Power Control Register, controls various power-related operations.
PWR_CSR: Power Control/Status Register, provides information about power status.

Clock Control Registers:

RCC_CR: Clock Control Register, controls the clock system.
RCC_CFGR: Clock Configuration Register, configures the clock system.
RCC_CIR: Clock Interrupt Register, controls clock interrupts.
RCC_AHB1ENR: AHB1 Peripheral Clock Enable Register, enables clock signals to AHB1 peripherals.
RCC_APB1ENR: APB1 Peripheral Clock Enable Register, enables clock signals to APB1 peripherals.
RCC_APB2ENR: APB2 Peripheral Clock Enable Register, enables clock signals to APB2 peripherals.

DMA (Direct Memory Access) Registers:

DMA_SxCR: DMA Stream Configuration Register, configures the DMA stream.
DMA_SxNDTR: DMA Stream Number of Data Register, specifies the number of data items to transfer.
DMA_SxPAR: DMA Stream Peripheral Address Register, holds the peripheral address for DMA transfer.
DMA_SxM0AR: DMA Stream Memory 0 Address Register, holds the memory address for DMA transfer

.

ARM: Advanced RISC Machines (the company that designs the ARM architecture)
Cortex-M4: The specific ARM Cortex-M series microcontroller core, which is designed for efficient and low-power embedded applications.
RISC: Reduced Instruction Set Computer, a type of microprocessor architecture that emphasizes simplicity and efficiency.
MCU: Microcontroller Unit, a compact integrated circuit that includes a microprocessor core, memory, and peripherals.
STM32: STMicroelectronics 32-bit microcontroller family.
F: Flash memory-based microcontroller.
446: Specific variant or model within the STM32F4 series.
RE: Reduced Embedding, which refers to a smaller package variant of the STM32F4 series.
GPIO: General Purpose Input/Output, refers to the pins on the microcontroller that

can be configured for various digital I/O operations.
RCC: Reset and Clock Control, a peripheral on STM32 microcontrollers responsible for configuring system clocks.
HSE: High-Speed External, an external crystal oscillator used as the main clock source in STM32 microcontrollers.
HSI: High-Speed Internal, an internal oscillator used as an alternative clock source in STM32 microcontrollers.
PLL: Phase-Locked Loop, a circuit that generates a high-frequency clock signal from a lower-frequency input.
AHB: Advanced High-Performance Bus, a bus that connects the CPU core to high-performance peripherals and memories.
APB: Advanced Peripheral Bus, a bus that connects the CPU core to lower-speed peripherals.
USART: Universal Synchronous/Asynchronous Receiver/Transmitter, a type of serial communication interface.
DMA: Direct Memory Access, a feature that allows data to be transferred between peripherals and memory without CPU intervention.
NVIC: Nested Vectored Interrupt Controller, a peripheral that manages interrupts in Cortex-M4 microcontrollers.
CMSIS: Cortex Microcontroller Software Interface Standard, a vendor-independent hardware abstraction layer for Cortex-M microcontrollers.
DSP: Digital Signal Processing, refers to the DSP capabilities of the Cortex-M4 core.
FPU: Floating-Point Unit, an optional hardware unit in Cortex-M4 that provides hardware support for floating-point operations.
EXTI: External Interrupt/Event Controller, a peripheral that handles external interrupts and events.
TIM: Timer, a peripheral used for general-purpose timing operations and generating PWM signals.
I2C: Inter-Integrated Circuit, a serial communication protocol used for connecting peripheral devices.
SPI: Serial Peripheral Interface, a synchronous serial communication protocol used for communication between microcontrollers and peripheral devices.
UART: Universal Asynchronous Receiver/Transmitter, a type of serial communication interface.
PWM: Pulse-Width Modulation, a technique for varying the duty cycle of a signal to control power delivered to a load.
DMA: Direct Memory Access, a feature that allows data to be transferred between peripherals and memory without CPU intervention.
ADC: Analog-to-Digital Converter, a peripheral used to convert analog signals into digital values.
DAC: Digital-to-Analog Converter, a peripheral used to convert digital values into analog signals.
USB: Universal Serial Bus, a popular interface for connecting devices to a computer or other host systems.
RTC: Real-Time Clock, a peripheral used for keeping track of time and generating interrupts at specified times.
CMSIS-DSP: Cortex Microcontroller Software Interface Standard for Digital Signal Processing, a collection of DSP functions and algorithms for Cortex-M4 processors.
TCM: Tightly-Coupled Memory, a type of fast and low-latency memory that is closely integrated with the CPU core in Cortex-M4 microcontrollers.
NVIC: Nested Vectored Interrupt Controller, a peripheral that manages and prioritizes interrupts in Cortex-M4 microcontrollers.
BPU: Breakpoint Unit, a debug feature that allows setting breakpoints and

monitoring program execution in Cortex-M4.
VFP: Vector Floating-Point, an optional extension to the FPU in Cortex-M4 that provides hardware support for vector floating-point operations.
ETM: Embedded Trace Macrocell, a debug and trace component that captures program execution information in Cortex-M4.

Jonogon things

Sunday, June 4, 2023 10:30 AM

Architecture

- Two operation states → (i) Thumb State
  (ii) Debug State
- Two operation modes → (i) Handler Mode
  (ii) Thread Mode
- Handler mode → always Privileged Access Level
- Thread mode → privileged / unprivileged Access Level
- By default, Cortex M4 processors start in Privileged Thread Mode and Thumb State.
- Register Bank in Cortex M4, M3 has 16 registers.
- R0 - R12 → General Purpose Registers

Stack Pointer (R13)

(i) Main Stack Pointer (MSP)

↳ Default stack pointer

→ Selected after Reset or when the processor is in handler mode

→ Initial value of MSP is taken from the first word of the memory during the reset sequence.

(ii) Process Stack Pointer (PSP)

↳ Can only be used in Thread Mode

↳ initial value of PSP is undefined.

↳ not necessary to use PSP if the application doesn't require an embedded OS.

- Both MSP and PSP are 32-bit, but the lowest 2 bits

require an embedded OS.

- Both MSP and PSP are 32-bit, but the lowest 2 bits of the SPs are always 0 and writes to these 2 bits are ignored.

Link Register (R14)

Used for holding the return address when calling a function or a subroutine.

When a function/ subroutine call is made, the value of LR is updated automatically. If a function is needed to call another function, it needs to save the value of LR in the stack first. Otherwise, the current value of LR will be lost when the function call is made.

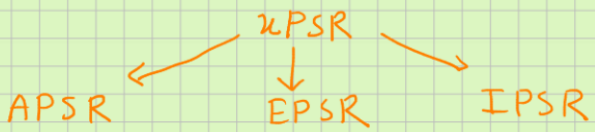
- Bit 0 of LR is Readable and Writeable
  Bit 0 = 1 to indicate Thumb State.

Program Counter (R15)

- Readable and Writeable
- Read returns current instruction address +4
- Write causes Branch operation

Special Registers

→ contain the Processon status and define the operation states and interrupt / exception masking.



APSR → Application Program Status Register
EPSR → Execution Program Status Register
IPSR → Interrupt Program status Register

N, Z, C, V Flags

Overflow flag will be set if

(i) Negative + Negative = Positive

(ii) Positive + Positive = Negative

(iii) Positive + Negative or Negative + Positive && $MSB_{in} \neq MSB_{out}$

Case (i)

```
  1000 0110
  1000 1000
-------------
1 0000 1110
```

Carry, C = 1
Negative, N = 0
Zero, Z = 0
Overflow, V = 1

Case (ii)

```
  0111 0101
  0100 0101
-------------
  1011 1010
```

C = 0
Z = 0
N = 1
V = 1

Case (iii)

```
  0110 0110
  1000 1110
-------------
  1111 0100
```

C = 0
Z = 0
N = 1
V = 0
$MSB_{in} = 0$
$MSB_{out} = 0$

```
  0110 0110
  1100 1110
-------------
1 0011 0100
```

C = 1
Z = 0
N = 0
V = 0
$MSB_{in} = 1$
$MSB_{out} = 1$

Four bit arithmetic

```
  1101
  1011
-------
 11000
```

C = 1
Z = 0
N = 1
V = 0

PRIMASK

① 1 bit wide interrupt mask register

② When set it blocks all exceptions / interrupts apart from NMI and HardFault.

③ It raises the current exception priority level to 0, which is the highest level for a programmable exception / interrupt.

④ Most common usage for PRIMASK is to disable all interrupts for a time critical process.
After the time critical process is completed, the PRIMASK needs to be cleared to re-enable interrupts.

To disable interrupts,

```
MOVS R0, #1
MSR PRIMASK, R0
```

To enable / allow all interrupts

```
MOVS R0, #0
MSR PRIMASK, R0
```

FAULTMASK

① It also blocks HardFault

② Raises the current exception priority level to -1

③ Can be used to bypass MPU and suppress bus fault.

④ Unlike PRIMASK, FAULTMASK is cleared automatically at exception return.

To disable interrupt

```
MOVS R0, #1
MSR FAULTMASK, R0
```

To allow interrupts

```
MOVS R0, #0
MSR FAULTMASK, R0
```

BASEPRI

① Masks exceptions / interrupts based on priority level

② When BASEPRI is set to 0, it is disabled.

③ When it is set to a non-zero value, it blocks exceptions that have the same / lower priority level, while still allowing exceptions with a higher priority level to be accepted by the processor.

To disable all interrupts with priority 0x60 - 0xFF

```
MOVS R0, #0x60
MSR BASEPRI, R0
```

To allow all interrupts

```
MOVS R0, #0x0
MSR BASEPRI, R0
```

MRS ⇒ Move to Register from Special Register
MSR ⇒ Move to special Register from Register
MRC ⇒ Move Register from coprocessor
MCR ⇒ Move Register to Coprocessor

Control Register

① Control register defines
   (i) Selection of stack pointer (MSP / PSP)
   (ii) Access level in Thread Mode (Privileged / Unprivileged)

② For Cortex-M4 with a FPU, one bit of the Control Register indicates if the current context uses the FPU or not.

③ Control Register can only be modified in the Privileged Access level and can be read in both privileged and unprivileged

access levels.

④ After reset the control register is 0. This means the Thread Mode uses the MSP and has Privileged Accesses.

Programs in Privileged Thread Mode can switch the stack pointer selection or switch unprivileged access level by writing to control. However, once nPRIV (control bit 0) is set, the program running in Thread can no longer access the control register.

nPRIV (bit 0)
- → 0 : default, privileged access level
- → 1 : Unprivileged Access Level

SPSEL (bit 1)
- → 0 : default, Thread Mode MSP
- → 1 : Thread Mode PSP

: Handler Mode bit always 0 and write to this bit is ignored.

FPCA (bit 2) → Floating Point Context Active
- → only available in Cortex M4
- → 0 : default,
  FPU has not been used in the current context and no need to save Floating Point Registers.
- → 1 : Current context has used floating point instructions and need to save floating point registers.

ARM7 has 7 operating modes (obviously ☺)

① Supervisor (SVC) : Entered with Reset and when a software Interrupt Instruction is executed

② FIQ : Entered when a High Priority interrupt is raised.

③ IRQ : Entered when a low priority interrupt is raised.

④ Abort : Used to handle memory access violations.

⑤ Undef : Used to handle undefined instructions

⑥ System : Privileged Mode using the same registers on User Mode

⑦ User : Mode under which most applications or OS tasks run.

Exceptions & Interrupts

Interrupt : An interrupt is an event that can be triggered by a peripheral device or a software request to temporarily pause the currently executing code and transfer control to a specific ISR.

Exception : An exception is more general term that refers to any event that causes a change in the normal flow of program execution.

Polling vs Interrupt

Polling : You pick up the phone every few seconds to check whether you are getting a call.

Interrupt : Do whatever you should do and pickup the phone when it rings.

Interrupt Service Routine

A special block of code that gets executed automatically by the CPU when a corresponding interrupt occurs.

Also known as Interrupt Handler or Exception Handler.

Interrupt Vector Table

The entry points of all interrupt service routines are stored in a specific table. This special table is called Interrupt Vector Table.

- The second word in the memory is a pointer to the reset_handler which is the starting memory address or entry point of the function Reset_Handler()

The Reset_Handler contains the program code which is executed when the processor is reset.

Typically, the reset handler performs some hardware initialization first and then calls the main function. Immediately after the processor is reset, the PC is initialized to this value.

- Calculate the address which holds the address of the ISR for interrupt n.

Address of pointer = 64 + 4 * n

Example: EXTI3_IRQn = 9

∴ Address of pointer = 64 + 4 * 9 = 100 = 0X64

Note: For arm cortex processors, the bit 0 of the destination address must always be 1. This is because ARM cortex processors only support Thumb Instruction Architecture.

Exception Number	Exception Type	Priority
1	Reset	-3
2	NMI	-2
3	HardFault	-1
4	MemManageFault	Programmable
5	Bus Fault	"
6	Usage Fault	"
11	SVC	"
12	Debug Monitor	"
14	Pend SV	"
15	Systick	"

Interrupt Servicing Sequence

1. Peripheral asserts interrupt request
2. Processor suspends currently operating task
3. Processor executes an ISR to service the peripheral and optionally clear the interrupt request.
4. Resume previously suspended task.

Interrupt Handling Sequence

1. Enter an Interrupt Handler
   - (i) HW saves PC
   - (ii) HW puts an interrupt/exception number in a register
   - (iii) HW switches to privileged interrupt handler mode
   - (iv) HW jumps to the PC specified in the interrupt vector table
2. ISR
   - (i) ISR saves/restores additional registers that will be used
   - (ii) ISR may disable interrupts while it is running
   - (iii) ISR finds out the reason for an interrupt and processes it.
   - (iv) ISR runs 'return-from-interrupt' instruction
3. Exit an interrupt handler
   - (i) HW restores HW saved registers
   - (ii) HW switches the privilege mode back
   - (iii) HW jumps to the saved PC.

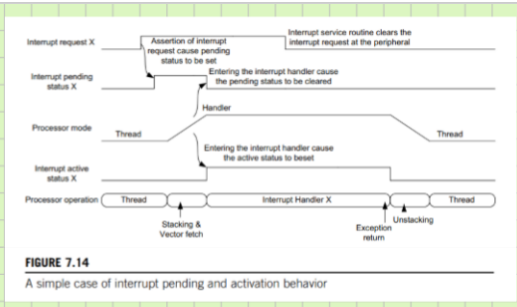
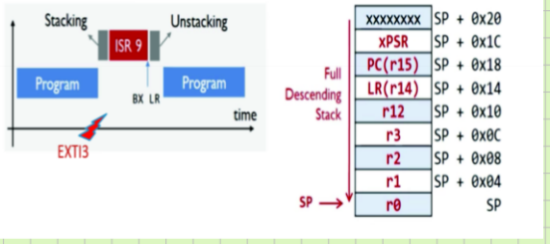
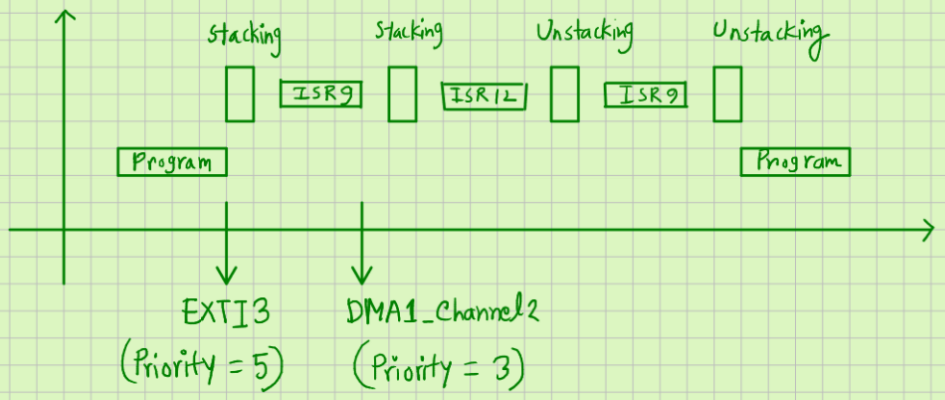


FIGURE 7.14
A simple case of interrupt pending and activation behavior

Single Interrupt Operation



Nested Interrupts (Example of Preemption)



Nested Interrupts (Tail Chaining)

When an exception takes place but the processor is handling another exception of the same or higher priority, the exception will enter the pending state.

When the processor finishes executing the current exception handler,

it can then proceed to process the pending exception/ interrupt request.

Instead of restoring the registers back from the stack (unstacking) and then pushing them onto the stack again (stacking), the processor skips the unstacking & stacking steps and enters the exception handler of the pended exception as soon as possible.

For a memory system with no-wait state, the tail-chain latency is only six cycles.

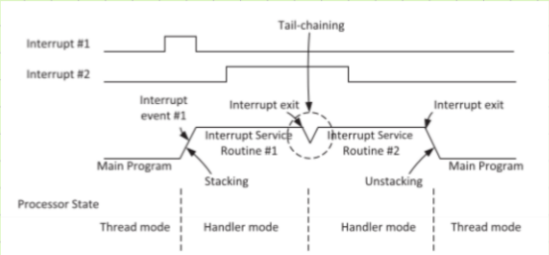
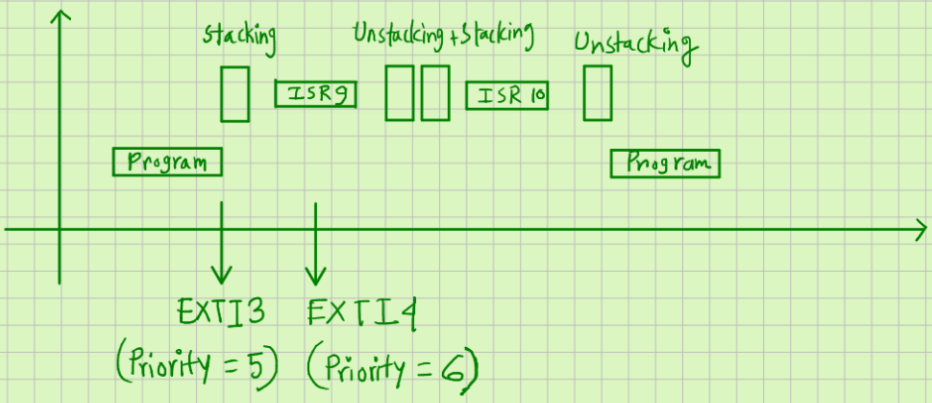


Figure 4: Tail Chaining



[1-15] : System Exceptions (Defined By ARM Core)
Rest 240 : Peripheral Interrupts (Defined By Chip Manufacturers)
Interrupt Number in PSR = Interrupt Number for CMSIS + 16
CMSIS => Cortex Microcontroller Software Interface Standard
NVIC => Nested Vector Interrupt Controller

How to enable the interrupt TIM7?

=> TIM7_IRQn = 49

NVIC → ISER [49/32] = 1 << (49 % 32);

or, NVIC → ISER [1] = 1 << 12;

Generalised Formula:

To enable,

```
NVIC → ISER [IRQn/32] = 1<< (IRQn % 32)
or, NVIC → ISER [IRQn >> 32] = 1 << (IRQn & 0x1F)
```

To disable,

```
NVIC → ICER [IRQn/32] = 1 << (IRQn % 32);
NVIC → ICER [IRQn >> 5] = 1 << (IRQn & 0x1F);
```

** Separating enable bits and disable bits in two separate sets of registers, ISER and ICER provides great convenience for software programmens. It allows us to enable or disable interrupt flexibly, without worrying about writing zero to the other bits in the target register.

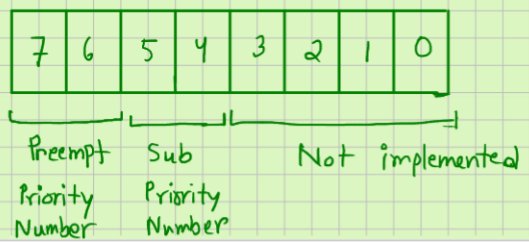
** fixed Priority for Reset, NMI and Hardfault.

Reset → Priority = -3; IRQn = Not applicable

NMI → Priority = -2; IRQn = -14

HardFault → Priority = -1; IRQn = -13

Interrupt Priority Register (IP)

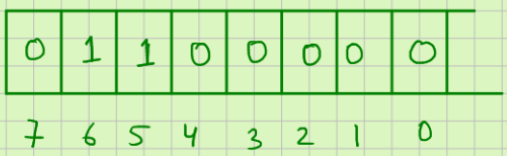


Preempt Priority determines if one interrupt can preempt another.

Sub priority determines which interrupt will be handled first who two interrupts of the same preempt priority arrive at the same time.

The sub priority is used only when two interrupts with the same preempt priority value are pending. Specifically, the interrupt with lower sub priority will be handled first.

NVIC_SetPriority (7,6);



IP = 0x60 = 96

Equivalent to NVIC->IP[7] = (6<<4) & 0xff;

NVIC_SetPriorityGrouping (n)

n	# of bits in Preempt	# of bit in Sub
0	0	4
1	1	3
2	2	2
3	3	1
4	4	0

This function will change the AIRCR register after performing a seqquence of unlocking process.

AIRCR -> Application Interrupt and Reset Control Register

NVIC Registers for Interrupt Control :-

(i) ICTR → Interrupt Controller Type Register (Read Only)
(ii) ISER → Interrupt Set Enable Register
(iii) ICER → Interrupt Clear Enable Register
(iv) ISPR → Interrupt Set Pending Register
(v) ICPR → Interrupt Clear Pending Register
(vi) IABR → Interrupt Active Bit Register (Read Only)
(vii) IPR → Interrupt Priority Bit Register

SCB (System Control Block) Registers for Interrupt Control :

(i) ICSR (Interrupt Control and Status Register)
→ set & clear the pending status of system exceptions including Systick, Pend SV, NMI
→ Determine the currently executing exception number

(ii) VTOR (Vector Table Offset Register)
→ Defines the starting address of the memory being used as the vector table.

(iii) AIRCR (Application Interrupt and Reset Control Register)
→ controls priority grouping
→ provides endianness info

(iv) SHP (System Handler Priority Register)

SVC → SHP [0] → MemManage Fault PL
[1] → BusFault PL
(8 bits, [2] → UsageFault PL [11] → Systick PL
reset value [7] → SVC PL
0) [8] → Debug Monitor PL
[0] → PendSV PL

(v) SHPCSR (System Handler Priority Control and State Register)
→ enables MemManage Fault, Usage fault, Bus fault

Special Registers:

- PRIMASK: used to disable all exceptions except NMI and Hard faults
  - To disable all interrupts:
    MOVS R0, #1
    MSR PRIMASK, R0
  - To allow all interrupts:
    MOVS R0, #0
    MSR PRIMASK, R0
- FAULTMASK: similar to the previous one except that it changes the effective current priority level to -1, only the NMI exception handler can works
  - To disable all interrupts:
    MOVS R0, #1
    MSR FAULTMASK, R0
  - To allow all interrupts:
    MOVS R0, #0
    MSR FAULTMASK, R0
- BASEPRI: disable interrupts with priority lower than a certain level
  - To disable all interrupts with priority 0x60-0xFF:
    MOVS R0, #0x60
    MSR BASRPRI, R0
  - To cancel the masking:
    MOVS R0, #0x0
    MSR BASEPRI, R0

Fault (in our stars?)

Watchdog Timer and BOD (Brown-Out Detector)

In many simple microcontrollers you can find features like a watchdog timer and BOD.
The watchdog can be programmed to trigger if the counter is not cleared within a certain time and can be used to generate a reset or NMI.

The BOD can be used to generate a reset if the supply voltage drops to a certain critical level.

When a failure occurs and the processor stops responding, it might take a bit of time for the watchdog to kick in.
For some safety critical applications, a 1msec delay can be a matter of life or death.

In order to allow problems to be detected as early as possible, the cortex-m processors have a fault exception mechanism included. If a fault is detected, a fault exception is triggered and one of the fault exception handlers is executed.

By default, all the faults trigger the HardFault exception.

Cortex M4 has 3 additional Configurable fault exception handlers:

(i) MemManage Fault
(ii) Bus Fault
(iii) Usage Fault

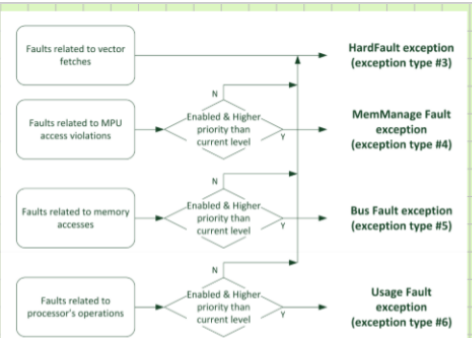


FIGURE 12.1
Fault exceptions available in ARMv7-M architecture

MemManage Faults

Caused by violation of access rules defined by MPU configurations

(i) Unprivileged tasks trying to access a memory region that is privileged access only.

(ii) Access to a memory location that is not defined by any defined MPU regions (except PPB which is always accessible by privileged code)

(iii) Writing to a memory location that is defined as read only by the MPU.

The MemManage Fault can also be triggered when trying to execute program code in execute Never (XN) region such as the PERIPHERAL, DEVICE or SYSTEM region.

The accesses could be data accesses during

(i) Program Execution
(ii) Program Fetches
(iii) stack operations during execution sequences

For instruction fetches that trigger a MemManage fault, the fault triggers only when the failed program location enters the execution stage.

For a MemManage Fault triggered by stack operating during exception sequence,

(i) If the MemManage fault occured during stack pushing in the exception entrance sequence, it's called a stacking error.

(ii) If the MemManage fault occured during stack popping in the exception exit sequence, it's called a unstacking error.

Bus Faults

The bus faults can be triggered by error responses from the Processor Bus Interface during a memony access.

For example, Instruction Fetch / Prefetch Abort
Data Read/Write / Data Abort

If a bus error is returned at vector fetch, the HardFault exception would be activated even when the Bus Fault exception is enabled.

If the bus error happened in the instruction fetch, the bus fault triggers only when the failed program location enters the execute region.

A memory system can return error responses if :

(i) The processor attempts to access an invalid memory location. In this case, the transfer is sent to a module in the bus system called Default slave.

(ii) The device is not ready to accept a transfer (for example, trying to access DRAM without initializing the DRAM controller)

(iii) The bus slave receiving the transfer request returns a error response. For example, it might happen if the transfer type/size is not supported by the bus slave.

(iv) Unprivileged access to the PPB that violates the default memory access permission.

The bus fault can also occur during stacking and unstacking of the exception handling sequence :

(i) If the bus error occurred during stack pushing in the exception entrance sequence, it's called a stacking error.
(i) If the bus error occurred during stack popping in the exception exit sequence, it's called a unstacking error.

Bus Faults can be classified as :

(i) Precise Bus Faults : Fault exceptions happened immediately when the memory access instruction is executed.

(ii) Imprecise Bus Faults : Fault exceptions happened sometime after the memory access instruction is executed.

The reason for a bus fault to be Imprecise is due to the presence of write buffers in the processor bus interface.

When the processor writes data to a bufferable address, the processor can proceed to execute the next instruction even if the transfer takes a number of clock cycles to complete.

The write buffer allows the processor to have higher performance but makes debugging a bit harder because by the time the bus fault exception is triggered, the processor could have executed a number of instructions, including branch instructions. If the branch target can be reached via several paths, it could be hard to tell where the faulting memory access took place unless you have an instruction trace. To help with debugging such situations, you can disable the write buffer using the DISDEFWBUF bit in Auxiliary Control Register.

Always Precise: Read operations & accesses to Strongly Order Region e.g. PPB

Usage fault

The Usage Fault exception can be caused by a wide range of factors:

(i) Execution of an undefined instruction (trying to execute FP instruction when FPU is disabled)

(ii) Execution of coprocessor instructions

(iii) Trying to switch to ARM state

(iv) Invalid EXC-RETURN code during exception-return sequence. For example, trying to return to Thread level with exceptions still active.

(v) Unaligned memory access with multiple load or store instructions

(vi) Execution of SVC when the priority level of SVC is the same or lower than current level.

(vii) Exception return with Interrupt-Continuable Instruction (ICI) bits in the unstacked xPSR, but the instruction being executed after exception return is not a multiple load/store instruction

(viii) It is also possible, by setting up the Configuration Control Register (CCR) to generate usage faults for the following

- Division By Zero
- All unaligned memory accesses

Please note that the floating point instructions supported by the Cortex-M4 are not co-processor instructions (e.g., MCR, MRC; see section 5.6.15). However, slightly confusingly, the register that enables the floating point unit is called the Coprocessor Access Control Register (CPACR; see section 9.10).

Hard Faults

The HardFault exception is triggered by escalation of configurable fault exceptions.

HardFault can also be triggered by:

(i) Bus error received during a vector fetch

(ii) Execution of BreakPoint Instruction (BKPT) with a debugger attached (halt debugging not enabled) and debug monitor exception is not enabled.

When reaching a printf operation, the processor executes a BKPT instruction and halt, and the debugger can detect the halt, check the register status and the immediate value in the BKPT instruction. Then the debugger can display the message or the character form the message in the printf statement. If the debugger is not attached, such operations result in HardFault and executes the HardFault exception handler.

Enabling the handlers:

① SCB → SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk

② SCB → SHCSR |= SCB_SHCSR_BUSFAULTENA_Msk

③ SCB → SHCSR |= SCB_SHCSR_USGFAULTENA_Msk

Table 12.1 Registers for Fault Status and Address Information

Address	Register	CMSIS-Core Symbol	Function
0xE000ED28	Configurable Fault Status Register	SCB->CFSR	Status information for Configurable faults
0xE000ED2C	HardFault Status Register	SCB->HFSR	Status for HardFault
0xE000ED30	Debug Fault Status Register	SCB->DFSR	Status for Debug events
0xE000ED34	MemManage Fault Address Register	SCB->MMFAR	If available, showing accessed address that triggered the MemManage fault
0xE000ED38	BusFault Address Register	SCB->BFAR	If available, showing accessed address that triggered the bus fault
0xE000ED3C	Auxiliary Fault Status Register	SCB->AFSR	Device-specific fault status

Table 12.2 Dividing Configurable Fault Status Register (SCB->CFSR) Into Three Parts

Address	Register	Size	Function
0xE000ED28	MemManage Fault Status Register (MMFSR)	Byte	Status information for MemManage Fault
0xE000ED29	Bus Fault Status Register (BFSR)	Byte	Status for Bus Fault
0xE000ED2A	Usage Fault Status Register (UFSR)	Halfword	Status for Usage Fault

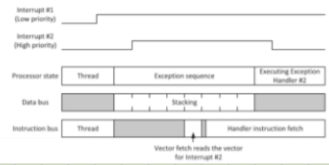
FIGURE 12.2
Configurable Fault Status Register partitioning

Late Arrival

When an exception takes place, the processor accepts the exception request and starts the stacking operation.

If during this stacking, another exception of higher priority takes place, the higher priority late arrival exception will be serviced first.

For example, if Exception #1 (lower priority) takes place a few cycles before Exception #2 (higher priority), the processor will behave as shown in following figure, such that Handler #2 is executed as soon as the stacking completes.



Lazy Stacking

Lazy stacking is a feature related to stacking of the registers in the floating point unit.

If the FPU is available and enabled, and if it has been used, the registers in the register bank of the floating point unit will contain the data that might need to be saved.

To stack the required floating point registers for each exceptions, system will need to carry out an additional 17 memory pushes each time, which will increase the interrupt latency from 12 to 29 cycles.

In order to reduce the interrupt latency, Cortex M4 processor has a feature called Lazy Stacking.

When an exception arrives with the floating point unit enabled and used, the longer stack frame is used.

The values of these floating point registers are not actually written into the stack frame. The lazy stacking mechanism only reserves the stack space for these registers, but only the R0-R3, R12, LR, Return Address, & PSR are stacked.
In this way, the interrupt latency remains at 12 clock cycles.

When the lazy stacking happens, an internal register called LSPACT is set and another 32-bit register called FPCAR (Floating Point Context Address Register) stores the address of the reserved stack space for the floating point register.

If the exception handler does not require any floating point operation, the floating point registers remain unchanged throughout the operation of exception handler and are not restored at exception exit.

If the exception handler does need floating point operations, the processor detects the conflict and stalls the processor, pushes the floating point registers into the reserved stack space and clears LSPACT. After that, the exception handler resumes. In this way, the floating point registers are stacked only if they are necessary.

When an interrupt request arrives during lazy stacking, the lazy stacking operation will be stopped and normal exception stacking starts.

Because the floating point instruction which triggered the lazy stacking has not yet been executed, the PC value stacked for the interrupt will point to the floating point instruction.

When the interrupt service completed, the executing return will return to this floating point instruction and reattempting this instruction will again trigger the lazy stacking operation.

Lecture 8

① Why is Bit-banding necessary ?

=> If a CPU needs to change the value of a bit and can only write a byte at a time, it must first read the current value into a temporary register, modify that value with a logical operation and then write the final result.
This three step process is named Read-Modify-Write.
Using Read-Modify-Write operations to set bits works fine when you're doing one thing at a time. Problems can arise when an application is doing multiple things concurrently.

If an interrupt occurs between the read and modify operations that changes the value in the register, the new value will get overwritten. This race-condition could lead to undesired behaviour.

Bit Banding is a special feature of Cortex M3/M4 processors where any bit stored between 0x20000000 and 0x20100000 in SRAM or 0x40000000 and 0x40100000 in the Peripheral Memory has a corresponding alias address which can be used to reference that bit specifically.

ARM cortex-M4 features a 1MB area in SRAM memory called Bit Band Region. In this region, each bit can be accessed individually.

To access a Bit Band Region bit you need to do so via an aliased region, where it maps each bit in that region to an entire word in a second memory region (Bit band alias region)

A write to a word in the alias region performs a write to the corresponding bit in the Bit Band Region.

Reading a word in the alias region will return the value of the corresponding bit in the Bit Band Region.

This operations take a single machine instruction thus eliminate race conditions. This is especially useful for interacting with Peripheral registers where it is often necessary to set and clear individual bits.

Formula & Solved Example

bit_word-offset = (byte offset x 32) + (bit number x 4)

bit_word-addr = bit-band-base + bit-word-offset

bit_word-offset => the position of the target bit in the bit band memory region.

bit-word_addr => the address of the word in the alias memory region that maps to the targeted bit

bit-band-base => the starting address of the alias region

byte-offset => the number of the byte in the bit band region that contains the targeted bit

bit-number => the bit position (0 to 7) of the targeted bit.

- Original start address of the stored variable: 0x20000004
- Base address of that bit band region (in SRAM): 0x20000000
- Base address of that bit band alias region (in SRAM): 0x22000000
- Total offset of stored variable from bit band region: 0x4
- Alias formula: (total offset * 0x20)+(number of bit you want to manipulate *4)= (0x4 * 0x20)+(3*4)=0x14
- Bit band alias memory location: 0x22000000+0x14=0x22000014
- So use 0x22000014 to manipulate the 3rd bit of memory location 0x20000004

SUCH A DISGRACE

Original start address of the stored variable: 0x20000004

Base address of that bit band region (in SRAM): 0x20000000

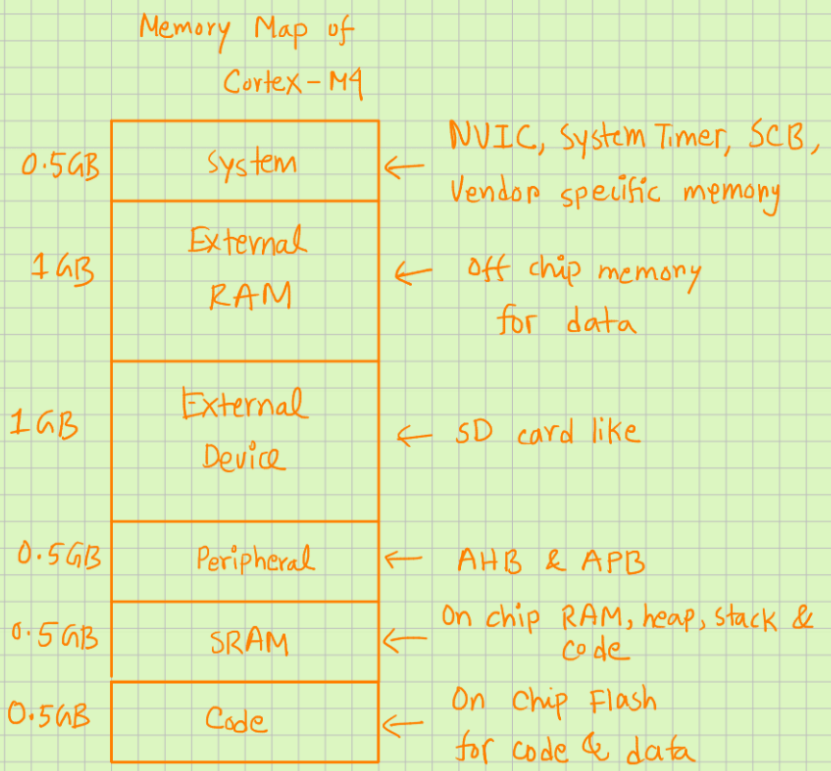
Base address of that bit band alias region (in SRAM): 0x22000000

Byte offset = 0x4

Bit word offset = (byte offset * 0x20)+(the bit position of the targeted bit *4)
= (0x4 * 0x20)+(3*4)=0x8C

Bit word address = 0x22000000+0x8C=0x2200008C

So use 0x22000014 to manipulate the 4th bit of memory location 0x20000004



Interrupt

Monday, 22 May, 2023 10:38 AM

Memory address of ARM cortex M4 has total of 32 bits.
Memory space divided into 6 different pre defined regions.

Region 1: Code = Use to store program code. Stores data as well.

↳ size is limited to half a gigabyte

Region 2: SRAM = Static Ram

↳ supports 0.5 GB

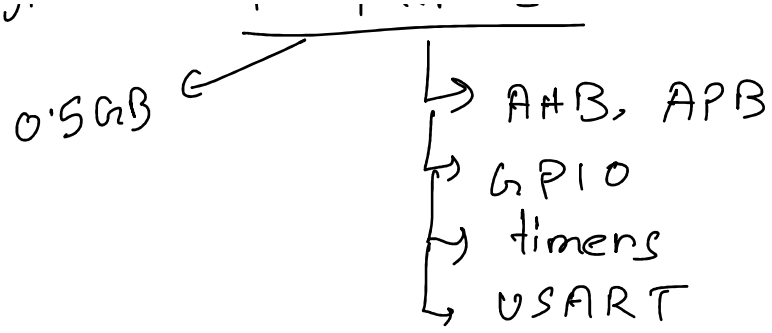
↳ Stores data

↳ heaps

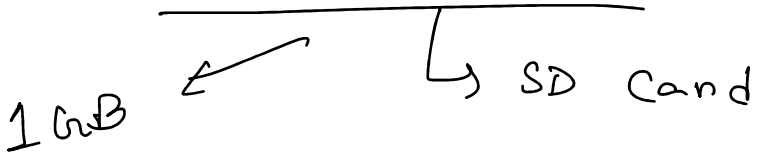
↳ stacks

Region 3: Peripherals

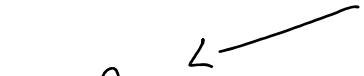
↳ APB AHB



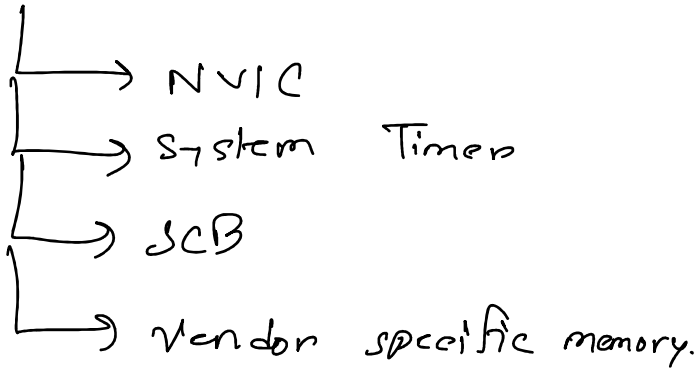
Region 4: External device:



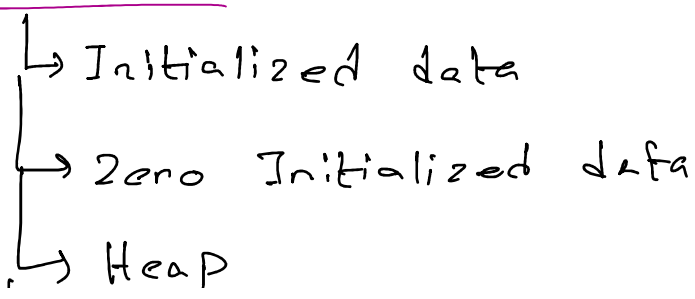
Region 5: External RAM: Off chip memory. Large Data block Storage.



Region 6: System

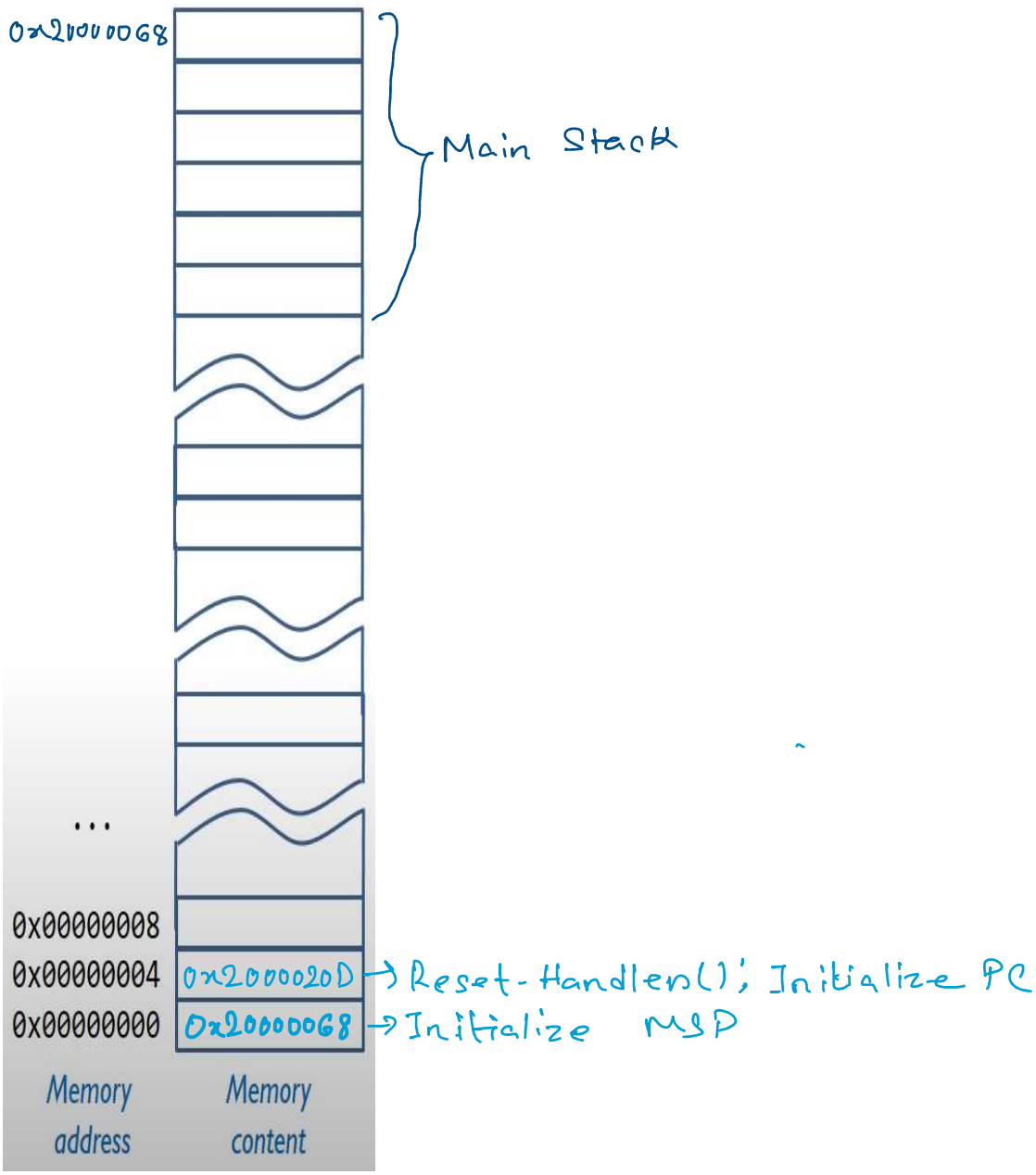


SRAM



- Heap
- Stack

Interrupt vector table 0 2



First word in the memory is Initialize MSP (main stack pointer) Main stack typically located in SRAM.

Processor যখন reset হয় Reset handler থেকে তার instruction code execute করে।

For ARM cortex M processors, if the interrupt number is (n), the pointer

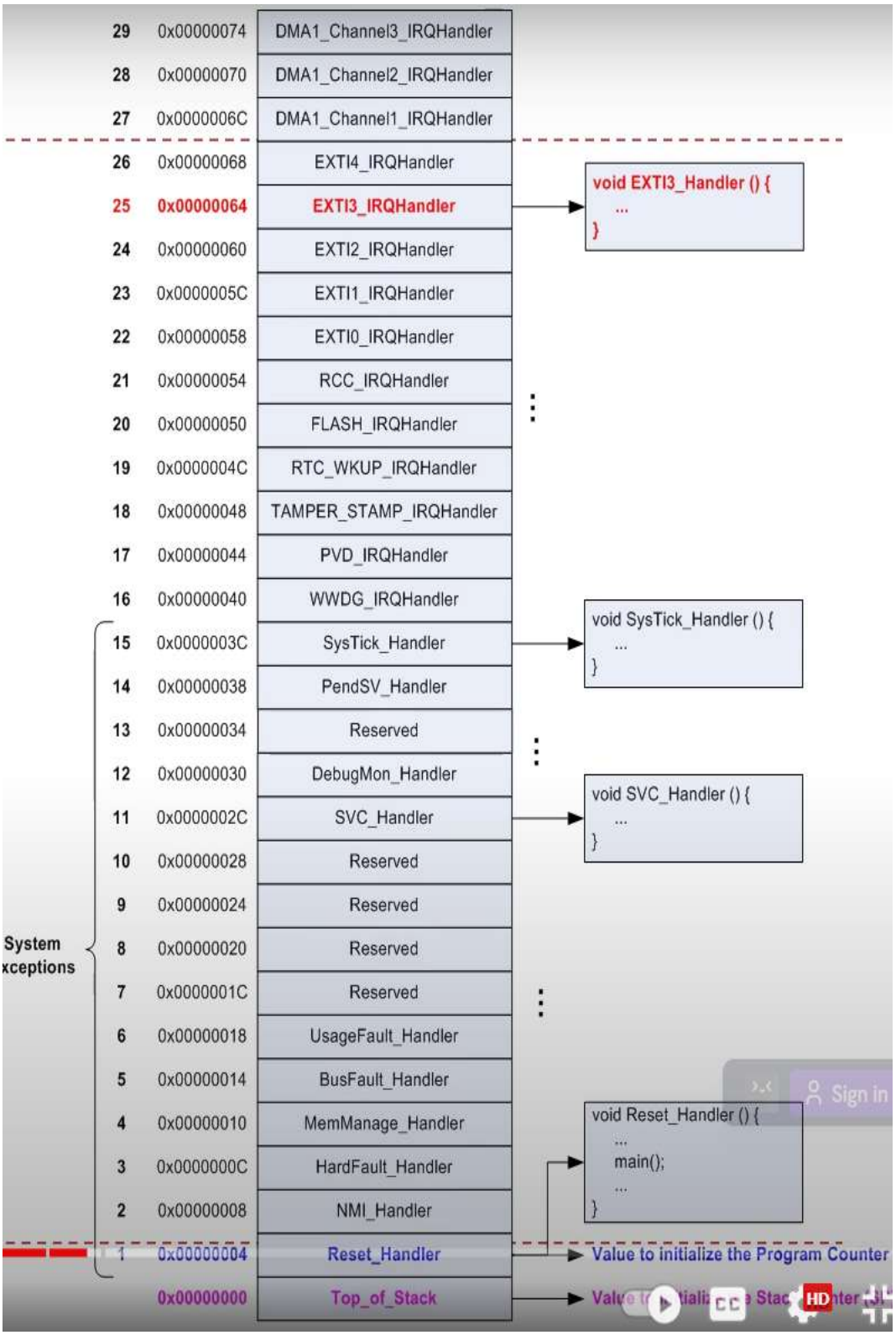
interrupt number is (n), the pointer that points the interrupt service routine is stored at:

Address of pointer = 64 + 4 * n

Suppose Interrupt number of EXTI3_IRQn = 9

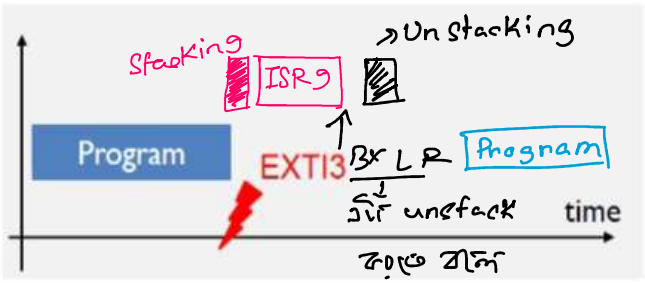
∴ Address of pointer to EXTI3 ISR
= 64 + 4 x 9
= 100
= 0x64

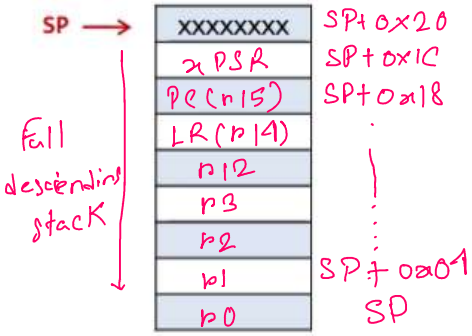
→ address in the vector table.



How NVIC handles an interrupt ?

Single Interrupt






```
I3_IRQHandler{
```

	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0 → 1	0
Priority Register	3	4	7	2	8

Suppose EXTI3 arrives at this time instant.
EXTI3 এর number হচ্ছে 9 (Predefined)

VIC first writes the Pending Register bit.
0 becomes 1 এটা করি

Now NVIC starts the stacking process.

FPU যদি use হয় তাহলে register stacked হয়।

Below is the order in which the SP or stack pointer pushes things.

① xPSR

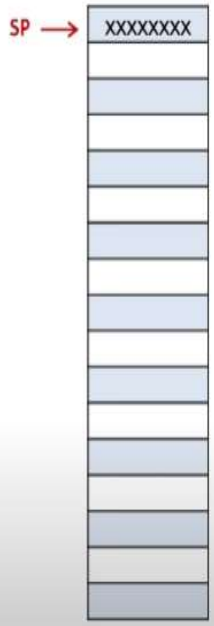
(5) PC (R15)
(3) LR (R14)
(4) R12
(5) R3
(6) R2
(7) R1
(8) R0

Then NVIC looks up the interrupt vector table, finds the starting address of the interrupt service routine of EXTI3.

এখন ~~[illegible]~~di: Pending Register এ value হয় 0 তখন Active এ value হয় 1.

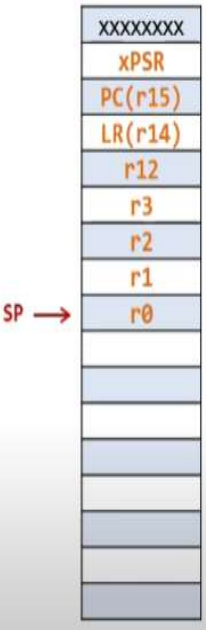
Multiple Interrupt:

Nested Interrupts: Example of Preemption



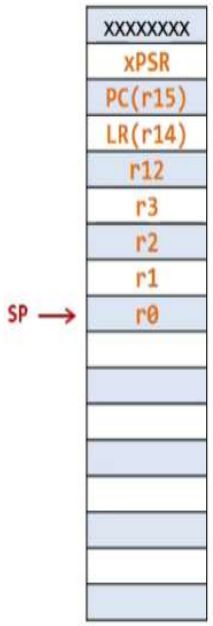
	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

Nested Interrupts: Example of Preemption



	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

Nested Interrupts: Example of Preemption

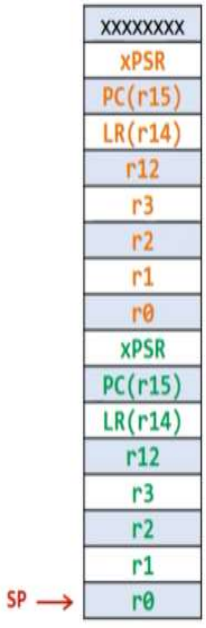


	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	1	0	0	0	0
Priority Register	3	4	7	5	3

36

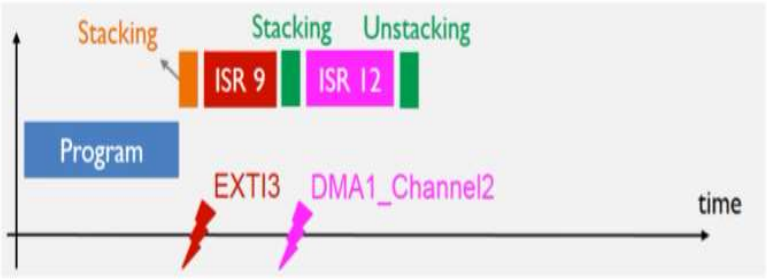
Lower priority value means higher urgency.

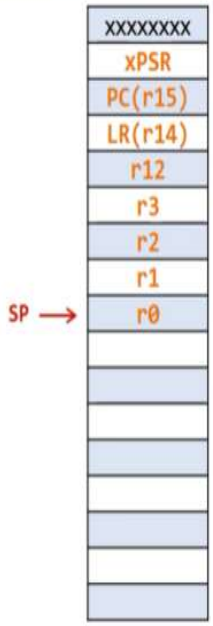
Nested Interrupts: Example of Preemption



	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	1	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

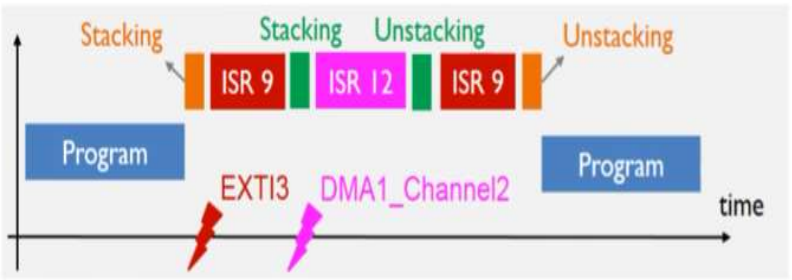
Nested Interrupts: Example of Preemption

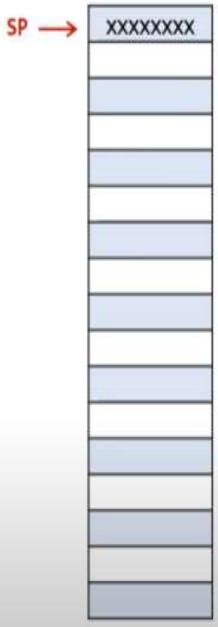




	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

Nested Interrupts: Example of Preemption

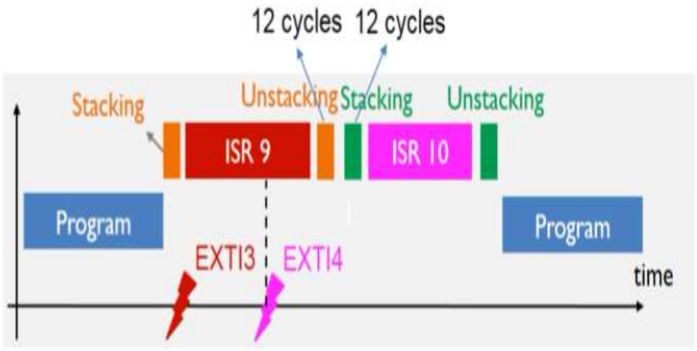


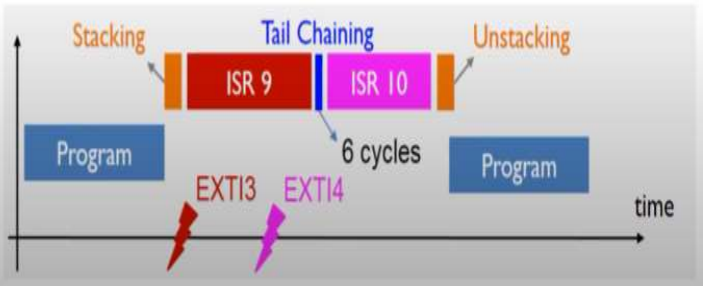


	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

2.10 Nested Interrupts: Tail Chaining

- EXTI3 → ISR 9
- EXTI4 → ISR 10
- Suppose EXTI4 has less urgency than EXTI3.
  - EXTI4 has a higher numeric priority value than EXTI3.





Interrupt Enable & Interrupt priority

Cortex M supports 256 interrupts

First 16, system interrupt (-16 → -1)

Later 240, Peripheral u (0 - 239)

CMSIS- Cortex Microcontroller Software Interface standard.

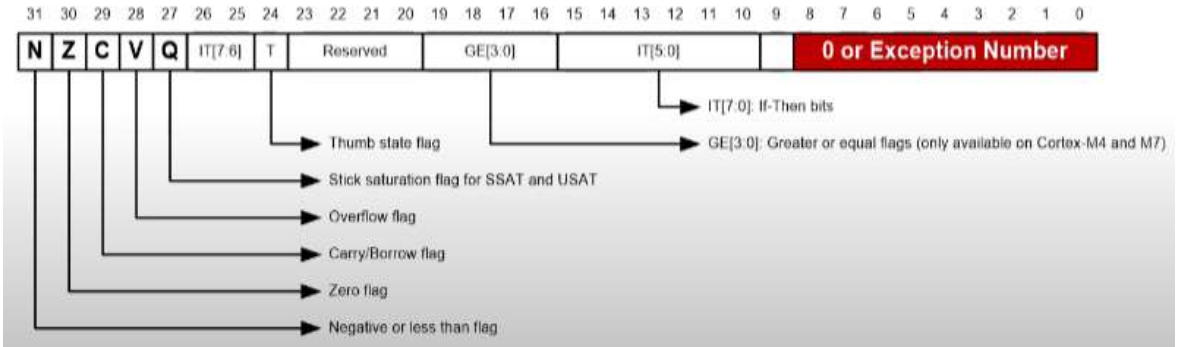
1.40

Interrupt Number in CMSIS *vs* in PSR

Interrupt number for CMSIS functions

```
NVIC_DisableIRQ (IRQn);              // Disable interrupt
NVIC_EnableIRQ (IRQn);               // Enable interrupt
NVIC_ClearingPending (IRQn);         // clear pending status
NVIC_SetPriority (IRQn, priority); // set priority level
```

Interrupt number in Program Status Register (PSR)



Interrupt Number in PSR = 16 + Interrupt Number for CMSIS

Enable an Interrupt

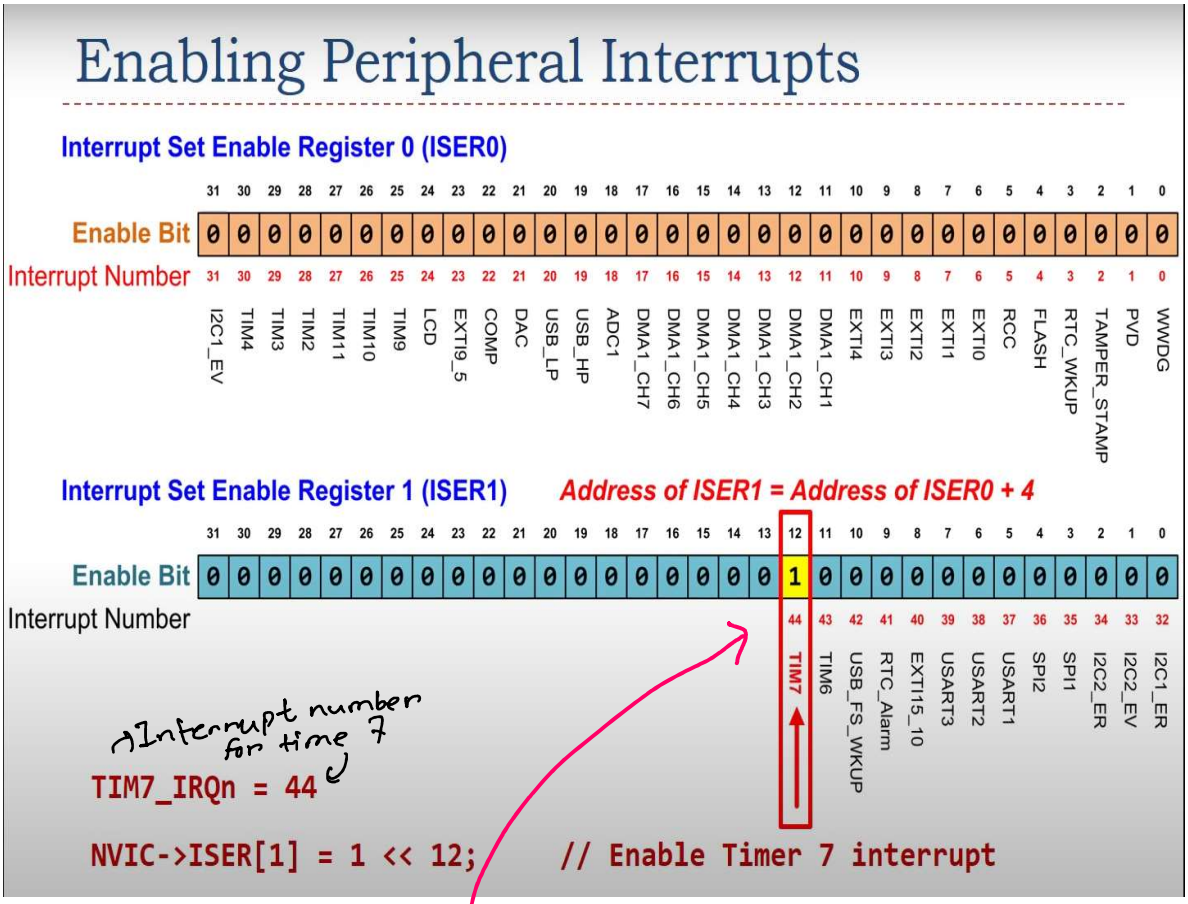
- Enable a system exception → RESET & Hardfault
  - Some are always enabled (cannot be disabled)
  - No centralized registers for enabling/disabling
  - Each are control by its corresponding components, such as SysTick module

- Enable a peripheral interrupt
  - Centralized register arrays for enabling/disabling
  - **ISER** registers for enabling – to enable peripheral intr
  - **ICER** registers for disabling – to disable " "

Interrupt set enable registers

↳ Interrupt set enable register

↳ " clear " "



ISER0 enables interrupt number 0→31

ISER1 " " " 32→63

To enable interrupt 44 we need to set bit 12 of ISER1 to 1.

To disable:

```
NVIC->ICER[1] = 1 << 12; // Disable Time 7
                         // interrupt
```

Each ISER registers has 32 bits.

Disable/Enable Peripheral Interrupts

- For all peripheral interrupts: `IRQn ≥ 0`
- Method 1:
  - `NVIC_EnableIRQ (IRQn);  // Enable interrupt`
  - `NVIC_DisableIRQ (IRQn); // Disable interrupt`
- Method 2:
  - Enable:
    - `NVIC->ISER[ IRQn / 32] = 1 << (IRQn % 32);`
    - Better solution:
    - `NVIC->ISER[ IRQn >> 5] = 1 << (IRQn & 0x1F);`
  - Disable:
    - `NVIC->ICER[ IRQn >> 5] = 1 << (IRQn & 0x1F);`

Interrupt Priority

- Inverse Relationship:
  - Lower priority value means higher urgency.
    - Priority of Interrupt A = 5,
    - Priority of Interrupt B = 2,
    - B has a higher priority/urgency than A.
- Fixed priority for Reset, HardFault, and NMI.

Exception	IRQn	Priority
Reset	N/A	-3 (the highest)
Non-maskable Interrupt (NMI)	-14	-2 (2nd highest)
Hard Fault	-13	-1

- Adjustable for all the other interrupts

hello

is interrupt a good thing?

is interrupt a good thing?

To enable interrupt 44 we'll have to do some magics.

Harvard Muddy 1

Sunday, 4 June, 2023 1:05 AM

Source Register

R5 1111 1111 0001 1100 0001 0000 1110 0111

LSL R0, R5, #7 : R0 = 1000 1110 0000 1000 0111 0011 1010000
7 bits

LSR R1, R5, #17 : R1 = 0000 0000 0000 0000 0111 1111 1000 1110

ASR R2, R5, #3 : R2 = 1111 1111 1110 0011 1000 0010 0001 1100

XOR R3, R5, #2F : 1110 0000 1000 0111 0011 1111 0111 1000

Sets flags if

N is negative N = 1

Z is zero Z = 1

C causes carry out C = 1

V is signed overflow V = 1

How to set condition flags:

Method 1 : Use CMP

CMP R5, R6

→ Does not save result

→ sets flags if result is 0 or neg

Method 2: use (S)

```
ADDS R1, R2, R3
```

Condition mnemonic:

```
CMP  R1, R2
SUBNE R3, R5, R8
```

SUB will execute if R1 ≠ R2

Unconditional branching:

```
MOV  R2, #17   ; R2 = 17
B    TARGET    ; branch to target
ORR  R1, R1, #0x4 ; not executed
TARGET
     SUB  R1, R1, #78  ; R1 = R1 + 78
```

This is a label

labels indicate instruction location

Branch not taken:

```
MOV  R0, #4        ;  R0 = 4
ADD  R1, R0, R0    ; R1 = R0 + R0 = 8
CMP  R0, R1        ; R0 - R1
BEQ  THERE         ; branch not taken (zz0)
```

```
CMP     R0, R1          ; R0 - R1
BEQ     THERE           ; branch not taken (zz0)
ORR     R1, R1, #1      ; R1 = R1 OR R1 = g

THERE
  - - - -
```

Some C code practice ?? — If statement

C Code	Assembly
`if (i == j)`	`; R0 = f, R1 = g, R2 = h,`
`  f = g + h`	`  R3 = i, R4 = j`
	`CMP  R3, R4`
`f = f - i ;`	`BNE  L1`
	`ADD  R0, R1, R2`
	`L1`
	`SUB  R0, R0, R3`
	OR we could do
	`CMP  R3, R4`
	`ADDEQ  R0, R1, R2`
	`SUB  R0, R0, R3`

If else statement

```
                        ; R0=f, R1=g, R2=h, R3=i,
if (i==j)               ; R4=j
  f=g+h;

                        CMP   R3, R4
else                    ADDEQ R0, R1, R2
  f=f-i;                SUBNE R0, R0, R2
```

while loops:

```
// determine the power
// of a such that
// 2^a=128

int pow=1;
int x=0;

while(pow!=128){
  pow=pow*2;
  x=x+1;
}
```

```
; R0=pow, R1=x
MOV   R0, #1 ; pow=1
MOV   R1, #0 ; x=0

WHILE
CMP   R0, #128
BEQ   DONE

LSL   R0, R0, #1 ; pow=pow*2
ADD   R1, R1, #1
B     WHILE


DONE
```

for loop's

```
int sum = 0

for(int i ; i != 20; i=i+1)
  sum = sum + 1
```

```
R0, i, R1 = sum
MOV  R0, #1
MOV  R1, #0

FOR
CMP  R0, #10
BEQ  DONE

ADD  R1, R1, R0
ADD  R0, R0, #1
B  FOR

DONE
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]     ; R2 = array(i)
  LSL  R2, R2, #3               ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]     ; array(i) = R2
  SUBS R1, R1, #1               ; i = i - 1
                                ; and set flags
  BPL  FOR                      ; if (i>=0) repeat loop
```

Harvard Muddy 2

Friday, 2 June, 2023 8:38 PM

Call function: branch and link

BL

Return from function: Move the link register to PC: MOV PC, LR

Arguments: R0 - R3

Return value: R0

C Code:

```
int main(){
  simple();
  a=b+c;
}

void simple(){
  return;
}
```

Assembly:

```
BL    simple
ADD   R4, R5, R6

SIMPLE  MOV PC, LR
```

Makes PC = LR

C Code 8

```
int main()
{
    int y;
    ---  --
    y = diffofsums (2, 3, 4, 5);
    ----
}

int diffofsums (int f, int g, int h, int i)
{
    int result;
    result = (f+g) - (h+i);
    return result;
}
```

Assembly:

```
; R4 = y
MAIN
    ----
    MOV  R0, #2    ; arg 0 = 2
    MOV  R1, #3    ; arg 1 = 3
    MOV  R2, #4    ; arg 2 = 4
    MOV  R3, #5    ; arg 3 = 5
```

```
BL   DIFFOFSUMS   ; call function
MOV  R4, R0       ; y = returned value
....
; R4 = result
DIFFOFSUMS
   ADD  R8, R0, R1  ; R8 = f+g
   ADD  R9, R2, R3  ; R9 = h+i
   SUB  R4, R8, R9  ; result = (f+g) - (h+i)
   MOV  R0, R4      ; put return value in R0
   MOV  PC, LR      ; return to caller.
```

Addressing modes:

Without offset:

```
LDR r1, [r0]   ; r0 holds the memory address
```

with offset:

```
LDR r1, [r0, #4]   ; pre-index
LDR r1, [r0], #4   ; post-index
LDR r1, [r0, #4]!  ; pre index with update.
```

LDR r1, [r0, #4]! ; pre index with update.

→ r1 ← memory. word [r0+4]

r0 remains unchanged

In post index the value is loaded from the address in the base register r0.

offset is added to the base memo address.

So, r0 এর value এবার change হবে,

offset value r0 এর মানের সাথে যোগ হবে।

base r0 থেকে যে value load হয়

(মানে r1 এ নিয়ে আসা হয়।

r1 ← memory. word [r0]

r0 ← r0 + 4

Pre index with update:

LDR r1, [r0, #4]!

r1 ← memory. word [r0+4]

r0 ← r0+4

C code:

```
int f1(int a, int b){
 int i,x;
x= (a+b)*(a-b);
for(i=0; i<a; i++)
  x= x+ f2(b+i);
  return x;
}
```

```
int f2(int p){
 int n;
 n= p+5;
 return n+p;
}
```

Assembly

```
;R0 = a , R1 = b, R4 = i, R5 = x
F1
PUSH {R4, R5, LR}
ADD  R5, R0, R1
SUB  R12, R0, R1
MUL  R5, R5, R12
MOV  R4, #0

FOR
CMP  R4, R0
BGE  RETURN
PUSH {R0, R1}
ADD  R0, R1, R4
BL   F2
ADD  R5, R5, R0
POP  {R0, R1}
ADD  R4, R4, #1
B    FOR

RETURN
MOV  R0, R5
POP {R1, R5, LR}
MOV  PC, LR
```

```
}                 POP {R1, R5, LR}
                  MOV PC, LR

                  ; R0 = P, R1 = n
                  F2
                  PUSH {R4}
                  ADD  R4, R0, #5
                  ADD  R0, R0, R4
                  POP  {R4}
                  MOV  PC, LR .
```

Recursive function call:

C Code:

```
int factorial(int n)
{
  if(n <= 1)
    return 1;

  else
    return (n*
    factorial(n-1));
```

Assembly language:

```
FACTORIAL   ; store R0 on stack
STR  R0, [SP, #-4]!
STR  LR, [SP, #-4]!   ; store LR on stack
CMP  R0, #2           ; if (n0 >= 2) then branch to else
BHS  ELSE
MOV  R0, #1           ; otherwise return 1
ADD  SP, SP, #8       ; restore SP 1
MOV  PC, LR
```

```
factorial(n-1));        MOV PC, LK
                        ELSE
                        SUB R0, R0, #1 -> n=n-1
}                       BL FACTORIAL
                        LDR LR, [SP], #4 -> restore LR
                        LDR R1, [SP], #4
                        MUL R0, R1, R0 -> R0 = n * factorial(n-1)
                        MOV PC, LR -> return
```

Binary Coding:

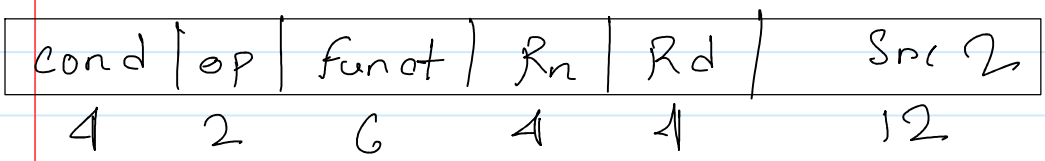
Data processing:

Operands:

- Rn = 1st source register
- Src2 = Source Register 2
- Rd = Destination Register

Control fields:

- cond : specifies conditional execution
- OP : operation code or opcode
- funct : function / operation

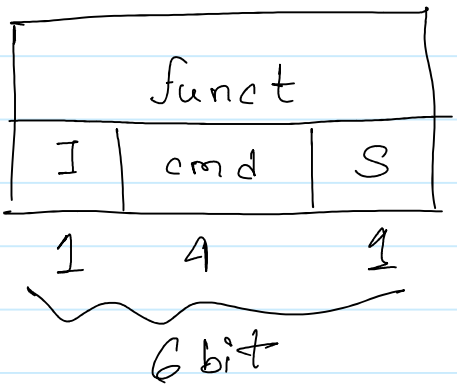


total 32

op = 00_2 for data processing instructions.

funct has 3 components

- cmd
- I bit
- S bit → Status bit → indicates whether the instruction updates the condition flag.



Cmd - Specifies the data processing instruction

Cmd = $(0100)_2$ for ADD

Cmd = $(0010)_2$ for SUB

I bit:

I = 0 ; if Src 2 is a register

I = 1 ; if Src2 is an immediate

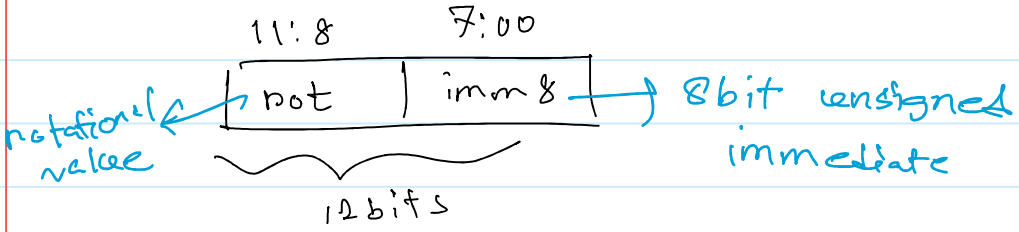
Src2 can be:

Immediate

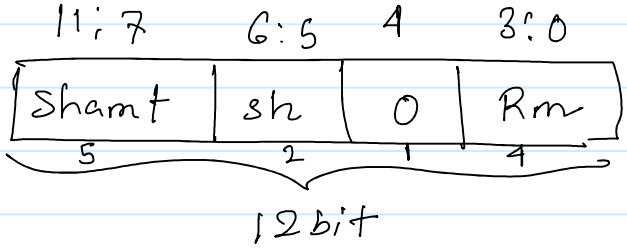
Register

Register shifted Register

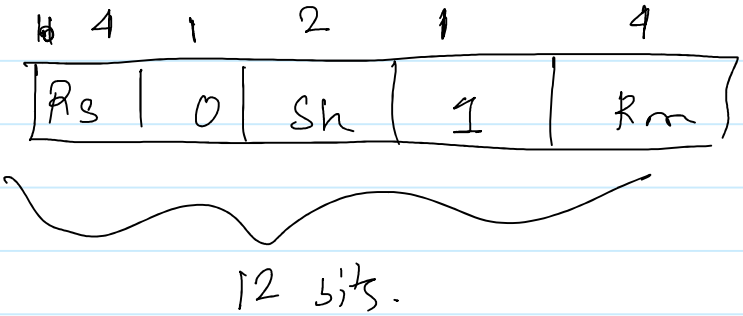
If I = 1, src_2 is an immediate.



If I = 0 and src2 is Register



If I = 0 and src2 is register shifted register



DP instruction with Immediate Src 2

ADD R0, R1, #42

cond	op	funct	Rn	Rd	Src2
4	2	6	4	4	12

cond = 1110_2 (unconditional execution)

op = $(00)_2$ (0) for data processing instructions.

cmd = $(0100)_2$ (4) for ADD

Src2 is an immediate, so I = 1

Rd = 0, Rn = 1

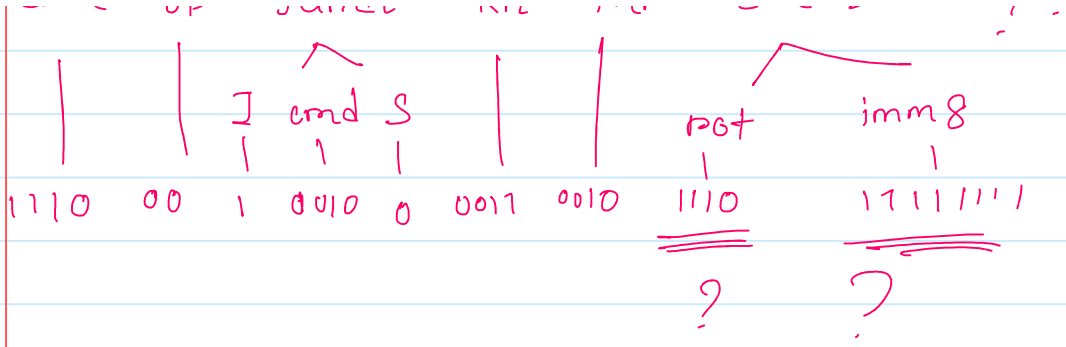
imm 8 = 42, rot = 0

cond	op	I	cmd	S	Rn	Rd	Shamt5	sh	Rm
1110	00	1	0100	0	0001	0000	0000	0101010	
E	2		8		1	0	0	2A	

E2810020

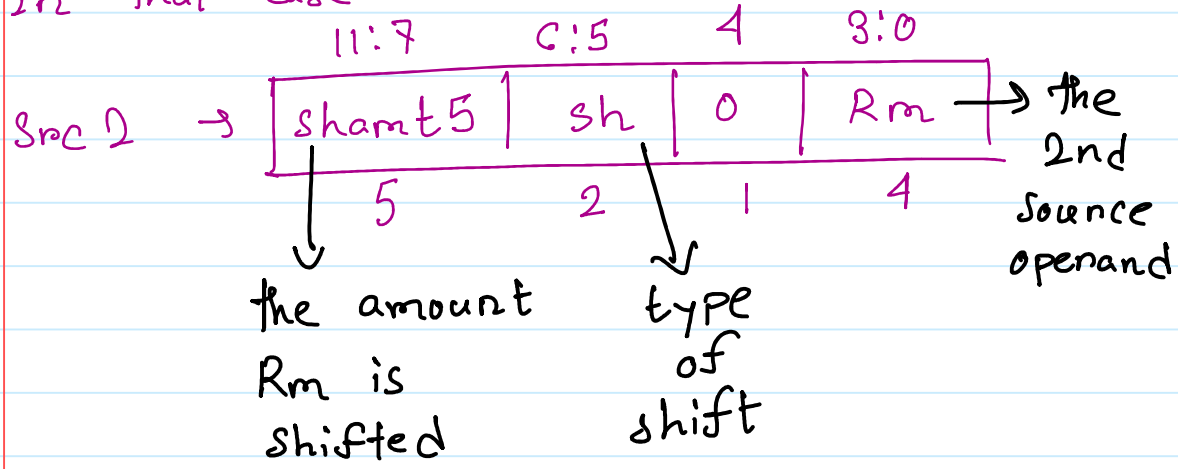
SUB R2, R3, #0xFF0

cond op funct Rn Rd Src2 ??



Src2 can be register.

In that case



unshifted version of Rm (shamt5 = 0, sh = 0)

5	2	1	4
✓			

ADD R5, R6, R7

cond	op	I	cmd	S	Rn	Rd	shamt	sh	0	Rm
1110	00	0	0100	0	0110	0101	00000	0	0	0111

Now consider shifted version:

ORR R9, R5, R3 LSR #2

Basically we want to do.

Basically we want to do.

R9 = R5 OR (R3 >> 2)

4	2	1	4	1	4	4	5	2	1	4
cond	op	I	cmd	S	Rn	Rd	shamt	sh	0	Rm
1110	00	0	1100	0	0101	1001	00010	01	0	0011

Now consider Register shifted Register:

EOR R8, R9, R10, ROR R12

Operation : R8 = R9 EOR (R10 ROR R12)

4	2	1	4	1	4	4					
cond	op	I	cmd	S	Rn	Rd	Rs	0	sh	1	Rm
1110	00	0	0001	0	1001	1000	1100	0	11	1	1010

Shift Instruction Encoding

LSL	00
LSR	01
ASR	10
ROR	11

Shift instruction: immediate shamt

এখানে main instruction টাই shift instruction okay? আমরা immediate value

ROR R1, R2, #23

operation → R1 = R2 ROR 3

cmd ⇒ 1101 for all shift op
↳ LSL, LSR, ASR, ROR

So, এক্ষেত্রে, Register আকারে যেমন src2 হেমন

5	2	1	4
src2 →	shamt	sh	0	Rm

cond	op	I	cmd	S	Rn	Rd	shamt	sh	0	Rm
1110	00	1	1101	0	~~0010~~ 0000	0001	10111	11	0	~~0010~~ 0010

কেন Rn=0 & Rm=2? R2

Shift Instruction: Register Shamt

আগেরটা ছিলো Immediate Shamt

ASR R5, R6, R10

Operation: R5 = R6 ASR R10

এখানে Register shifted Register হবেনা।

In that case Src_2 = Rs 0 sh 1 Rm

cond	op	I	cmd	S	Rn	Rd	Rs	0	sh	1	Rm
1110	00	0	1101	0	~~0110~~	0101	1010	0	10	1	0110

0000

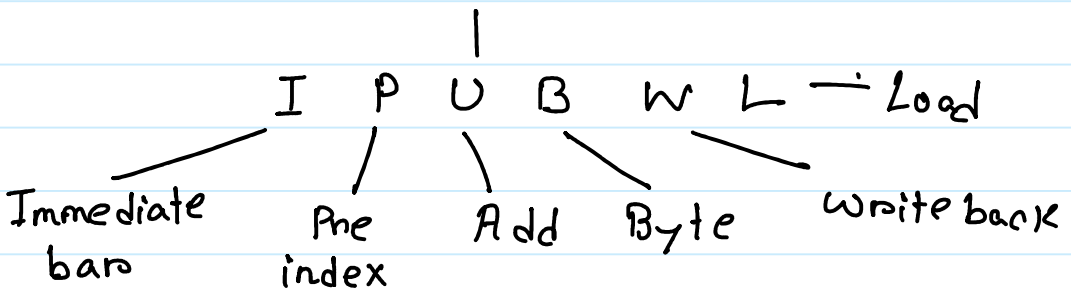
Memory Instructions:

Data processing ↳ default op value = 00
Memory " " " " = 01

Rn = base register
Rd = Source (store), destination (load)
Src2 = offset
funct = 6 control bits.

Simple:

cond op funct Rn Rd Src2



Address = Base address + offset

LDR R1, [R2, #4]

Base address = R2 , offset = 4
Address = (R2 + 4)

Base address always in a register

FPU

Sunday, 4 June, 2023 12:34 AM

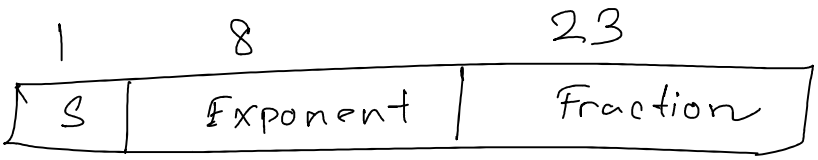
Floating point

FPU is optional. সবগুলো contex M4 এর মধ্যে FPU নাই।

Contex M4 = Contex M3 + DSP + FPU (optional)

Contex M4 এর FPU supports single precision only.

Single precision means floating point of 32 bit.



ARM Loops YouTube

Friday, 2 June, 2023 10:52 AM

while loop:

```
while (t0 < 100){
  t0 = t0+1;
}
```

ARM code

```
CMP   t0, #100
BGE   endLoop
ADD   t0, t0, #1
B     startLoop
endLoop
```

For loop:

```
int i;
for(i=1; i<= 10; i++){
  sum = sum + i;
}
```

ARM code:

```
MOV   R0, #1;   i= 1
startLoop
CMP   R0, #6
BGT   endLoop
ADD   R1, R1, R0
ADD   R0, R0, #1   ; i=i+1
B     startloop
endLoop
```

Instruction Set

Sunday, 26 February, 2023 5:58 PM

ARM7TDMI has two operating states.

ARM	Thumb
32 bit word aligned arm	16-bit half word alligned arm

Transition between ARM and Thumb states does not affect the processor modes or register contents.

ARM7TDMI has two instruction sets

- 32 bit ARM instruction set
- 16 bit Thumb " "

In comparison with 16bit architectures, 32 bit architectures exhibit higher performance when manipulating 32 bit data, can address a large address space much more efficiently.

16 bit architecture has higher code density but less performance.

Thumb instructions are each 16 bits.

Have a corresponding 32-bit ARM instruction that has same effect on the processor model.

16bit thumb instructions are transparently decompressed to full 32 bit ARM instructions in real time.

The ARM7TDMI is bi-endian

- big endian
- little endian both
  - traditionally default for ARM processors.

ARM7TDMI supports following data types:

(i) word. 32 bit
(ii) half word 16 bit
(iii) byte 8 bit

Five addressing modes!

(i) Shifter operands for data

(i) Shifter operands for data processing instructions.

(ii) Load and store word or unsigned byte.

(iii) Load and store half word or load signed byte.

(iv) Load & store multiple

(v) Load & store coprocesson.

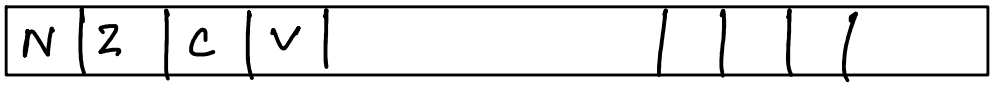
ARM has 37 registers in total

- 1 dedicated program counter
- 1 " current program status register
- 5 " saved " " "
- 30 general purpose registers.

r13 - Stack Pointer
r14 - Link Register
r15 - Program Counter

Accessing registers:

All instruction can access r0-r14 directly Most instructions also allow the use of PC.



CPSR → 32bit register

Current Program Status Register :

↳ always accessible in any processor mode

CPSR contains the following fields :

1. M[4:0] (Processor Mode) : Indicates current operating mode of the processor.

2. T (Thumb State) : This bit indicates whether the processor is currently in ARM state or Thumb state.
   T = 0 ARM state
   T = 1 Thumb state.

3. F (FIQ disable) :
   1 - FIQ disabled
   0 - " enabled

4. I (IRQ disable) :
   1 = IRQ disabled
   0 = " enabled

5. V (Overflow flag) :

6. C (Carry flag):

7. Z (Zero flag): bit is set when result of an arithmetic operation is

8. N (Negative Flag): when result is negative.

The ARM Instruction Set

ARM book

Instruction set encoding:

PRIMASK:

→ Single bit register that controls the priority of the interrupts.

⇒ When PRIMASK bit is set to 1 all the interrupts are set to be disabled except NMI (Non maskable interrupt) & Hard Fault Exceptions.

Suppose a critical section of code that must avoid interrupt.

Suppose a critical section of code that must avoid interrupt.

```
__disable_irq(); // Set PRIMASK
                 //bit to 1 to disable
                 //interrupts

// Code to execute without interrupt

__enable_irq()
```

Fault mask:

=> 1 bit register that allows the processor to ignore certain exceptions.

=> If the bit is set to 1, ignores everything except NMI & Hardfault.

=> If 0, exceptions are enabled.

```
__disable_fault_irq();
  // Code to handle
__enable_fault_irq();
```

BASE PRI:

=> 8 bit register that set the minimum priority level for interrupts.

→ When the BASEPRI value is set to greater than 0, all interrupts with a priority level equal to or lower than BASEPRI vaule are disabled.

```
uint32_t old_basepri = _get_BASEPRI();
_set_BASEPRI(5); //set minimum
                 //priority level to 5
_set_BASEPRI(old_BASEPRI)
```

In ARM Cortex-M microcontrollers, the nPRIV (non-privileged) bit is a control bit in the Control Register (or "cpsr" in Cortex-M4) that indicates whether the current code is executing in a privileged or non-privileged mode. Here's a more detailed explanation of the nPRIV bit:

1. Privileged Mode: In a privileged mode, the processor has access to all resources and registers in the system. This mode is typically used for kernel-level operations, such as device drivers and system calls. When executing in privileged mode, the nPRIV bit is cleared to 0.
2. Non-Privileged Mode: In a non-privileged mode, the processor has limited access to resources and registers in the system. This mode is typically used for user-level operations, such as application code and user space. When executing in non-privileged mode, the nPRIV bit is set to 1.
3. Switching Modes: The processor can switch between privileged and non-privileged modes using the "MSR" (Move to System Register) and "MRS" (Move from System Register) instructions. To switch from privileged to non-privileged mode, the processor must set the nPRIV bit to 1 using the "MSR" instruction. To switch from non-privileged to privileged mode, the processor must clear the nPRIV bit to 0 using the "MSR" instruction.

Overall, the nPRIV bit is an important control bit in ARM Cortex-M microcontrollers that allows for the separation of privileged and non-privileged code execution. By setting the nPRIV bit to 1 in non-privileged mode, the processor can prevent user-level code from accessing sensitive system resources and registers, improving system security and stability.

CONTROL register

The CONTROL register (Figure 4.9) defines:

- The selection of stack pointer (Main Stack Point/Process Stack Pointer)
- Access level in Thread mode (Privileged/Unprivileged)

Table 4.3 Bit Fields in CONTROL Register

Bit	Function
nPRIV (bit 0)	Defines the privileged level in Thread mode: When this bit is 0 (default), it is privileged level when in Thread mode. When this bit is 1, it is unprivileged when in Thread mode. In Handler mode, the processor is always in privileged access level.
SPSEL (bit 1)	Defines the Stack Pointer selection: When this bit is 0 (default), Thread mode uses Main Stack Pointer (MSP). When this bit is 1, Thread mode uses Process Stack Pointer (PSP). In Handler mode, this bit is always 0 and write to this bit is ignored.

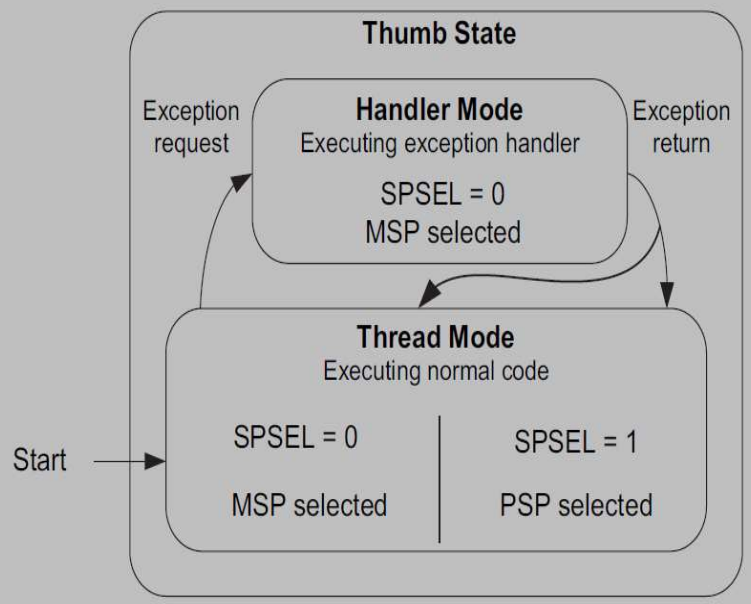


FIGURE 4.10

Stack Pointer selection

Addressing Modes:

① Immediate Addressing: Operand is specified directly in the instruction.

```
MOV  R1, #5
ADD  R2, R1, #3
```

② Register Addressing: Operand specified by register.

```
MOV  R2, R1
ADD  R3, R1, R2
```

③ Direct Addressing: Operand specified by memory address.

```
LDR  R1, =1000  ; Load the memory address 1000 into register r1.

LDR  R2, [R1]   ; load the contents of memory address stored in register r1 into R2.
```

① Indirect Addressing: Operand specified indirectly through a register that contains a memory address.

```
LDR  R1, =1000  → memory address
LDR  R2, [R1]
LDR  R3, [R2]
```

LDR R3, [R2]

⑤ Indexed addressing : Operand specified indirectly through a register that stores a memory address plus offset.

```
LDR R1, =1000 ; Load memory address
                1000 in R1
LDR R2, [R1, #4] ; R2 = content of
                   (R1 + 4)
```

ARM Book things.

Every instruction contains 4 bit condition code field in bits 31-28

1110 AL (Unconditional)

```
        B       label               ; branch unconditionally to label
        BCC     label               ; branch to label if carry flag is clear
        BEQ     label               ; branch to label if zero flag is set
        MOV     PC, #0              ; R15 = 0, branch to location zero
        BL      func                ; subroutine call to function
func
        ...
        ...
        MOV     PC, LR              ; R15=R14, return to instruction after the BL
        MOV     LR, PC              ; store the address of the instruction
                                    ; after the next one into R14 ready to return
        LDR     PC, =func           ; load a 32-bit value into the program counter
```

ARM has 16 data processing instructions.

Most data processing instructions take two source operands. MOV & MOVN take only one. Compare & Test instructions only update the condition flags. Other data processing instructions store a result to a register.

16 data-processing instructions, shown in Table

Mnemonic	Operation
AND	Logical AND
EOR	Logical Exclusive OR
SUB	Subtract
RSB	Reverse Subtract
ADD	Add
ADC	Add with Carry
SBC	Subtract with Carry
RSC	Reverse Subtract with Carry
TST	Test
TEQ	Test Equivalence
CMP	Compare
CMN	Compare Negated
ORR	Logical (inclusive) OR
MOV	Move
BIC	Bit Clear
MNV	Move Not

1 Instruction encoding

⟨*opcode1*⟩{⟨*cond*⟩}{S} ⟨*Rd*⟩, ⟨*shifter operand*⟩
⟨*opcode1*⟩ := MOV | MVN

⟨*opcode2*⟩{⟨*cond*⟩} ⟨*Rn*⟩, ⟨*shifter operand*⟩
⟨*opcode2*⟩ := CMP | CMN | TST | TEQ

⟨*opcode3*⟩{⟨*cond*⟩}{S} ⟨*Rd*⟩, ⟨*Rn*⟩, ⟨*shifter operand*⟩
⟨*opcode3*⟩ := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR

I bit [25] Distinguishes between the immediate and register fo

S bit [20] Signifies that the instruction updates the condition

Rn [19:16] Specifies the first source operand register.

Rd [15:12] Specifies the destination register.

shifter operand [11:0] Specifies the second source operand. *processing operands* for details of the shifter operands.

ARM has two classes of multiply instructions.

1. normal 32 bit
2. long, 64 bit

All multiply instructions take two register operands as the input to the multiplier.

The MLA (Multiply with Accumulate) instruction is used in ARM assembly language to multiply two registers and add the result to a third register. Here's an example of how to use the MLA instruction:

sql Copy code

```
MLA R0, R1, R2, R3 ; Multiplies the values in R1 and R2, adds the result to the val
```

In this example, the values in R1 and R2 are multiplied, and the result is added to the value in R3. The final result is then stored in R0. This instruction can be used to perform a multiplication and addition in a single operation, which can be more efficient than using separate instructions for each operation.

Examples

```
MUL     R4, R2, R1              ; Set R4 to value of R2 multiplied by R1
MULS    R4, R2, R1              ; R4 = R2 x R1, set N and Z flags
MLA     R7, R8, R9, R3          ; R7 = R8 x R9 + R3
SMULL   R4, R8, R2, R3          ; R4 = bits 0 to 31 of R2 x R3
                                ; R8 = bits 32 to 63 of R2 x R3
UMULL   R6, R8, R0, R1          ; R8, R6 = R0 x R1
UMLAL   R5, R8, R0, R1          ; R8, R5 = R1 + R8, R5
```

4.7.3 List of status register access i

MRS Move PSR to General-purpose Register.
MSR Move General-purpose Register to PSR.

Base Mode:

1) Register Base Mode

```
LDR R0, [R1]
; Load the value from the address
  stored in R1 into R0
```

2) Immediate base mode

```
LDR R0, =0x1234
; Load immediate value 0x1234
```

(3) PC relative base mode :

Load the value from the address located 4 bytes after the current instruction into R0

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

(4) Base register with immediate offset:

```
LDR R0, [R1, #8]
; Load the value from the adress
  stored in R1 plus 8 into R0.
```

(5) Base register with register offset :

```
LDR R0, [R1, R2]
```

Load the value from the address

stored in R1 + the value stored in R2 into R0.

Offset modes:

① Immediate offset:

```
LDR   R0, [R1, #4]!
```

Load the value from the address stored in R1 plus 4 into R0, and increments R1 by 4.

② Negative immediate address mode:

```
STR   R0, [R1, #-4]
```

Store the value in R0 at the address stored in R1 minus 4.

③ Scaled register offset mode:

```
LDR   R0, [R1, R2, LSL #2]
```

csharp Copy code

```
; Assume that R1 contains the base memory address
; Assume that R2 contains the offset value (in this case, 3)
; Assume that the value at memory address R1+12 (R1+(3<<2)) is 0x5678
MOV R1, #0x1000 ; Set R1 to the base memory address
MOV R2, #3 ; Set R2 to the offset value
LDR R0, [R1, R2, LSL #2] ; Load the value at memory address R1+12 into R0
```

In this example, the value 0x1000 is loaded into R1, which is the base memory address. The value 3 is loaded into R2, which is the offset value that needs to be scaled. The LDR instruction with base register with scaled register offset mode is then used to load the value from the address stored in R1 plus the value stored in R2 left-shifted by 2 (R1+(3<<2) in this case) into R0. Since the value at memory address 0x100C (R1+12) is 0x5678, the final value of R0 will be 0x5678.

It's important to note that the shift operand in this addressing mode specifies the amount to shift the offset value before adding it to the base register value to get the final memory address to load from. In this example, the shift operand is LSL #2, which means that the offset value stored in R2 is left-shifted by 2 bits (multiplied by 4) before it is added to the value stored in R1 to get the final m[illegible]rom.

Regenerate response

④ Base register with scaled immediate offset mode!

LDR R0, [R1, #8*4]

Load the value from the address stored in R1 plus 32 (8 times 4) into R0

Integer Status flags:

Table 4.6 ALU Flags on the Cortex-M Processors

Flag	Descriptions
N (bit 31)	Set to bit[31] of the result of the executed instruction. When it is "1," the result has a negative value (when interpreted as a signed integer). When it is "0," the result has a positive value or equal zero.
Z (bit 30)	Set to "1" if the result of the executed instruction is zero. It can also be set to "1" after a compare instruction is executed if the two values are the same.
C (bit 29)	Carry flag of the result. For unsigned addition, this bit is set to "1" if an unsigned overflow occurred. For unsigned subtract operations, this bit is the inverse of the borrow output status. This bit is also updated by shift and rotate operations.
V (bit 28)	Overflow of the result. For signed addition or subtraction, this bit is set to "1" if a signed overflow occurred.

Table 4.7 ALU Flags Example

Operation	Results, Flags
0x70000000 + 0x70000000	Result = 0xE0000000, N= 1, Z=0, C = 0, V = 1
0x90000000 + 0x90000000	Result = 0x30000000, N= 0, Z=0, C = 1, V = 1
0x80000000 + 0x80000000	Result = 0x00000000, N= 0, Z=1, C = 1, V = 1
0x00001234 – 0x00001000	Result = 0x00000234, N= 0, Z=0, C = 1, V = 0
0x00000004 – 0x00000005	Result = 0xFFFFFFFF, N= 1, Z=0, C = 0, V = 0
0xFFFFFFFF – 0xFFFFFFFC	Result = 0x00000003, N= 0, Z=0, C = 1, V = 0
0x80000005 – 0x80000004	Result = 0x00000001, N= 0, Z=0, C = 1, V = 0
0x70000000 – 0xF0000000	Result = 0x80000000, N= 1, Z=0, C = 0, V = 1
0xA0000000 – 0xA0000000	Result = 0x00000000, N= 0, Z=1, C = 1, V = 0

Chapter 5

Definitive ARM

Definitive ARM

5.1

ARM7TDMI came in 1995

Thumb provides subset of ARM instructions.

5.3 Understanding the Assembly language syntax

Instruction formatting:

```
label
mnemonic   operand1, operand2 ..
```

→ For data processing instructions first operand is the destination register of the operation

→ For memory read instruction first operand is the which the data is loaded into.

→ memory write এর memory তে যেটা

⇒ memory write হলে memory তে কোথায় লেখা হবে first operand সেটা hold করবে।

⇒ MOV
- → transfers data between registers
- → puts immediate constant into the register

Table 5.2 Commonly Used Directives	
Directive (GNU assembler equivalent)	**ARM Assembler**
THUMB (.thumb)	Specify assembly code as Thumb instruction in Unified Assembly Language (UAL) format.
CODE16 (.code 16)	Specify assembly code as Thumb instruction in legacy pre-UAL syntax.
AREA <section_name>{,<attr>}{,attr}... (.section <section_name>)	Instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.
SPACE <*num of bytes*> (.zero <*num of bytes*>)	Reserves a block of memory and fills it with zeros.
FILL <*num of bytes*>{, <*value*>{, <*value_sizes*>}} (.fill <*num of bytes*>{, <*value*>{, <*value_sizes*>}})	Reserves a block of memory and fills it with the specified value. The size of the value can be byte, half-word, or word, specified by value_sizes (1/2/4).
ALIGN {<*expr*>{,<*offset*>{,<*pad*>{,<*padsize*>}}}} (.align <alignment>{,<fill>{,<max}}})	Aligns the current location to a specified boundary by padding with zeros or NOP instructions. E.g., ALIGN 8 ; make sure the next instruction or ; data is aligned to 8 byte boundary
EXPORT <*symbol*> (.global <symbol>)	Declare a symbol that can be used by the linker to resolve symbol references in separate object or library files.
IMPORT <*symbol*>	Declare a symbol reference in separate object or library files that is to be resolved by linker.
LTORG (.pool)	Instructs the assembler to assemble the current literal pool immediately. Literal pool contains data such as constant values for LDR pseudo instruction.

For the Cortex-M3/M4 processors, a data processing instruction can optionally update the APSR (flags). If using the Unified Assembly Language (UAL) syntax, we can specify if the APSR update should be carried out or not. For example, when moving a data from one register to another, it is possible to use

```
MOVS R0, R1   ; Move R1 into R0 and update APSR
```

Or

```
MOV  R0, R1 ; Move R1 into R0, and not update APSR
```

The second type of suffix is for conditional execution of instructions. The Cortex-M3 and Cortex-M4 processors support conditional branches, as well as conditional execution of instructions by putting the conditional instructions in an IF-THEN (IT) instruction block. By updating the APSR using data operations, or instructions like test (TST) or compare (CMP), the program flow can be controlled based on conditions of operation results.

Table 5.3 Suffixes for Cortex-M Assembly Language

Suffixes	Descriptions
S	Update APSR (Application Program Status Register, such as Carry, Overflow, Zero and Negative flags); for example: ADDS R0, R1 ; this ADD operation will update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Conditional execution. EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. On the Cortex-M processors these conditions can be applied to conditional branches; for example: BEQ label ; Branch to label if previous operation result in ; equal status or conditionally executed instructions (see IF-THEN instruction in section 5.6.9); for example: ADDEQ R0, R1, R2 ; Carry out the add operation if the previous ; operation results in equal status

Function call:

function call involves branching to subroutine & then returning from it.

BL (Branch with Link)
BX LR (Branch exchang Link Register)

1. Simple function call:

```
AREA data, DATA, READWRITE

AREA myfunctions, CODE_READONLY

function1
    ADD R0. R0. #5
    BX LR
    ENDP

ENTRY
EXPORT MAIN
main
    MOV R0, #10
    BL function1 ; Call function1
                   and add5 to R0

Stop B stop
    END
```

Definitive guide 5.6

5.6.1 Moving data within processor

Table 5.4 Instructions for Transferring Data within the Processor

Instruction	Dest	Source	Operations
MOV	R4, ←	R0	; Copy value from R0 to R4
MOVS	R4, ←	R0	; Copy value from R0 to R4 with APSR (flags) update
MRS	R7, ←	PRIMASK	; Copy value of PRIMASK (special register) to R7
MSR	CONTROL,	R2	; Copy value of R2 into CONTROL (special register)
MOV	R3, →	#0x34	; Set R3 value to 0x34
MOVS	R3, →	#0x34	; Set R3 value to 0x34 with APSR update
MOVW	R6,	#0x1234	; Set R6 to a 16-bit constant 0x1234
MOVT	R6,	#0x8765	; Set the upper 16-bit of R6 to 0x8765
MVN	R3,	R7	; Move negative value of R7 into R3

MRS- Move to Register from Special Register

MSR- Move to special register from Reg.

DC D- Define Constant Doubleword

DCD যেকোন value ধরতে।

↳ 32bit address allocate করে।

PC or program counter holds the

PC or program counter holds the value of next instruction that is to be executed.

Offset is a constant value. It can be positive or negative.

LDR R0, [PC, #offset]

↳ PC-relative addressing is used here.

5.6.2 Memory access Instructions

Load - Read from memory
Store - Write to "

Immediate offset pre index is a type of addressing mode where the effective address is calculated by adding or subtracting an immediate value to the base register before loading or storing data.

① LDR R0, [R1, #4]!

R1 contains the base address. Adds 4 to R1, loads the word at

the new address into R0, and updates R1 with the new address.

(2) STR R2, [R3, #-8]!

R3 contains the base address. Subtracts 8 from R3. stores the value of R2 at the new address and updates R3 with the new address.

POST Index:

In post index the base address register is updated after the data transfer.

(1) LDR R0, [R1], #4

R1 contains the base address is loads the word at the address in R1 into R0. then adds 4 to R1 to update it.

(2) STR R2, [R3], #-8

R3 contains the base address stores the value of R2 at the address in R3. then subtracts 8 from R3 to update it.

Some loop things:

① Add numbers from 1 to 10.

```
AREA myCode, CODE, READONLY
ENTRY
EXPORT MAIN
main
    MOV R1, #0 ; R1 is used to store sum.
    MOV R2, #1 ; R2 is loop counter

loop
  ADD R1, R1, R2 ; Add loop counter
                   to the sum R1
  ADD R2, R2, #1 ; Add 1 to the loop
                   counter.
  CMP R2, #11
  BLT loop

stop B stop
END
```

Factorial of a given number:

Microcontroller

Saturday, 3 June, 2023 3:51 PM

Peripheral generally a port.
Not literally.

UART - Asynchronous

I2C, SPI - Synchronous

ADC ~~is~~ analog input ~~is~~

DAC a u output u

Clock configuration:

RCC_CR, RCC_CFGR, RCC_PLLCFGR,
RCC_APB1ENR, PWR_CR

Learning to ~~enable~~ configure clock:

1. Enable the desired clock source:

→ multiple clock sources are supported

HSI
↓
High Speed

HSE
High Speed External

PLL
→ Phase Locked Loop

& High speed External > Phase Locked Loop
High speed Internal

3) Enable the chosen clock source by configuring the corresponding bits in the RCC registers.

Enabling High Speed Internal clock source:

// Enable HSI clock ← Starts the HSI oscillator

```
RCC -> CR |= RCC_CR_HSION;
```

// Wait for HSI clock to stabilize

```
while (!(RCC -> CR & RCC_CR_HSIRDY));
```

HSI Ready

while loop টি চালু হয়ে যতক্ষণ না RCC_CR 1 হয়।

2. Configure the PLL (if applicable):

→ Must determine desired PLL input clock frequency & desired PLL

output freq.

→ PLL - Phase lock Loop

Multiplication করে input & output clock frequency এর,

→ Same enable & wait for stabilize.

```
RCC -> CR |= RCC_CR_PLLON;
while(!(RCC ->CR & RCC_CR_PLLRDY));
```

3. Configure Flash memory latency:

```
FLASH -> ACR = flash_acr;
```

4. Configure Prescalar & divider:

Prescalar হলো divider হলো একটা circuit, that divides the frequency of an input clock signal to generate lower frequency output signal.

Suppose input clock freq = 80 MHz

Prescalar/divider value = 4

Output clock frequency = 80/4 = 20 MHz

5. Select the system clock source:

↳ SYSCLK কোনটা হবে?

↳ Modify RCC configuration.

How to configure clock system on STM32F446 microcontroller.

1. Choose desired clock source from available options. Like: HSI, HSE. Maybe frequency 8MHz.

2. Enable the chosen crystal and wait for it to become ready.

For HSI, enable HSION.

then wait for HSIRDY bit to indicate readiness.

3. We need power to clock & peripherals.

So, PWREN. Enable PWREN bit

[illegible] PWREN.

in RCC-APB1ENR.

4. PWR-CR → main voltage baby
   11 → maximum voltage

5. FLASH-ACR → this register configures flash memory.

6. Configure prescalar for the buses :
   APB1, APB2, AHB1.
   by setting appropriate values in the RCC-CFGR Register.

7. Choose 로직 IN directly clock use 로직 for PLL utilization.

8. PLL use 로직 configure the main PLL (M, N, P)

9. Enable PLL by setting PLLON bit

10. Select desired clock source.

Wait System Clock ...

```
void SystemClock_Config(void);
{
  // Enable HSE
    RCC->CR |= RCC_CR_HSEON;
  while(!(RCC->CR & RCC_CR_HSERDY))
  {
      // wait for HSE to become ready
  }

// Enable power interface clock
  RCC->APB1ENR |= RCC_APB1ENR_
                  PWREN;

// Set voltage scaling to highest
// performance range.
    PWR->CR |= PWR_CR_VOS;

// Configure Flash memory latency
  FLASH->ACR |= FLASH_ACR_LATENCY_5WS;

// Configure AHB, APB1, APB2 prescalers
                  ↳ AHB prescaler no division
```

// Configure [illegible]

↳ AHB prescaler no division

```
RCC->CFGR |= RCC_CFGR_HPRE_DIV1;
Rcc->CFGR |= RCC_CFGR_PPRE1_DIV1;   // APB1
RCC->CFGR |= RCC_CFGR_PPRE2_DIV1;   // APB2

// Configure the main PLL
Rcc->PLLCFGR = (8 ---.

// Enable PLL
RCC->CR |= RCC_CR_PLLON;
while (!(RCC->CR & RCC_CR_PLLRDY))
{

}

// Select PLL for system clock source
RCC->CFGR |= RCC_CFGR_SW_PLL;
while{

}

}
```

IP

Input is 8 MHz
↳ goes to HSE
↳ goes to be selected by multiplexer

PLLM = 1/4
PLLN = 180
PLLP = 1/2

So, input X PLLM X PLLN X PLLP = ~~18~~

$8_{MHz} \times \frac{1}{4} \times 180 \times \frac{1}{2} = 180\ MHz$

∴ AHB = Prescalar = 1

∴ System clock is running at 180 MHz

APB1 prescalar = $\frac{1}{4}$

APB2 prescalar = $\frac{1}{2}$

Ques: APB1 ko max freq 45
APB2 " " " 90

[∴ Output = Input X PLLM X PLLN X PLLP]

Timer:

→ Counts clock pulses

→ Uses freq of internal clock

→ generates delay

→ After configuable number of counts, resets & and start from zero.

Timer

- Advanced control ↓ TIM1, TIM8
- General purpose
- Basic TIM6, TIM7 ↓ Capable of DMA

Short এ registers:

সবার name এ "TIMx_" থাকা must.

then comes the suffix.

- CR1 ↓ Control Register 1
- PSC ↓ Prescalar
- ARR ↓ Auto Reload Register

SR	CNT	DIER
↓	↓	↓
Status Register	Counter	DMA Interrupt Enable Register

Procnedure:

1. Choose minimum time delay on unit for the timer.

2. Enable the timer
   Provide it with clock
   How?
   use RCC_APB2ENR &
   TIM2_EN

3. Set prescalar - কেন? - to get desired delay.

বাসায় করে Prescalar বের করা:

$$P = F * D - 1$$

D = Desired delay in seconds
F = clock frequency in Hz
P = Required prescalar.

Suppose we want delay of 1 μs
clock frequency 30MHz

Then prescalar?

$$P = F * D - 1$$
$$= 90 \times 10^6 \times 1 \times 10^{-6} - 1$$
$$= 90 - 1$$
$$= 89$$

↳ use TIMx_ARR

4. Set the ARR → Auto Reload Register.
   ↳ Why? - for the maximum value the timer should count to.

5. Enable or turn on the counter
   ↳ use TIMx_CR1, CEN

6. Now, in order to count to a certain value, set the count to 0 and wait.
   ↳ use TIMx_CNT

Timer Configuration:

Time choose করার আগে step checking যেটা peripheral এ আমরা connected.

Datasheet এ দেখতে হবে কোন আমরা TIM6, TIM7 APB1 এ আছে, ↳ এর max

↳ 42 max frequency 45MHz

First enable the timer clock

** Always check datasheet numbers. not memorized info.

```
RCC -> APB1ENR |= (1 << 4)
```

Now, set the prescalar & the ARR

APB1 clock & APB1 timer has different frequencies.

$$\frac{90\,MHz}{90} = 10^6 = 1\ Mega\ second$$

Bits [15:0] = PSC[15:0] = Prescalar value.

Now use ARR,
So,

```
TIM6 -> PSC = 89
TIM6 -> ARR = 0xffff
```

Enable the timer, wait for the update flag

```
TIM6 -> CR1 |= (1<<0); <- enabling

while (!(TIM6 -> SR & (1<<0)); <- waiting
```

```
while (!(TIMG->SR & (1<<0)); // waiting
```

GPIO:

GPIO Ports A ... H

4 modes

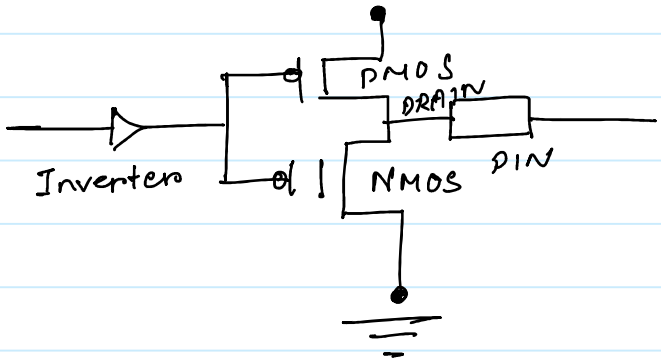
- 00 : Reset
- 01 : General purpose output
- 10 : Alternate function
- 11 : Analog mode

Input & Output मोड्स के different states:

Input states: → Floating (Input + no PUPD)
- Pull up / Pull down
- analog

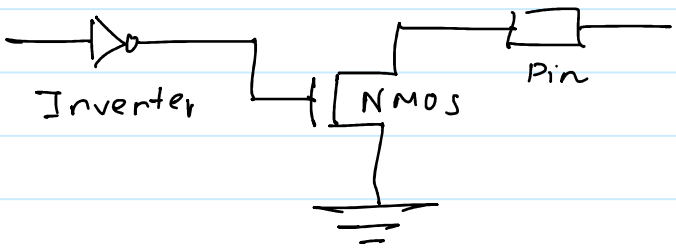
Output state: → push-pull
- Open drain / pull up / pull down

Push pull:



Signal 1 (High) → Vcc output & NMOS inactive
" 0 " → NMOS active

Open drain?



Signal 0 → output is pulled to ground
" 1 → " floating

Registers: "GPIOx_"

MODER	OTYPER	OSPEEDR
↓	↓	↓
Mode Register	Output type register	Output speed Register
PUPDR	IDR	ODR
↓	↓	↓

PUPDR	IDR	ODR
↓	↓	↓
Pull up Pull down Register	Input data Register	Output data Register

Procedure : Procedure :

1. Choose the pins'

2. Enable GPIO Port.
   Use:
   RCC_AHB1ENR,
   GPIOx_EN

3. Configure mode:
   ↳ GPIOx_MODER → Input
   → Output
   → Alternate
   → Analog

4. For output type, configure output type.
   push, pull and so on. OTYPER

5. Configure output speed OSPEEDR

6.

GPIO

Wednesday, 17 May, 2023 11:57 PM

General Purpose Input Output

GPIO as Output:

We want to set PA5 as output pin. let's say it's connected to an LED. we want to turn the LED ON/OFF. নিচে এমন একটি code:

```
// Enable GPIOA clock
RCC -> AHB1ENR |= RCC_AHB1ENR_
                  GPIOAEN;
```

Enables the clock for GPIOA. The AHB1ENR register is the part of Reset & Clock Control (RCC). The GPIOAEN is the bit within this register is set to enable the clock for GPIOA.

```
// Configure PA5 as output
GPIOA -> MODER |= (0b01 << (5*2));
// Sets the 5th pin to output mode
```

→ GPIOx_MODER Register is a part of the GPIO peripheral & used to define the functionality of GPIO pins.

00 : Input mode (reset)
01 : General purpose output mode
10 : Alternate function mode.
11 : Analog mode.

For each GPIO port the MODER register has 2 bits per pin - lowest 2 bits corresponding to pin (0) & highest 2 bits to pin (15)

→ This line sets the MODE for GPIOA pin PA5 to "output register".

The 0b01 is shifted left by 10 bits because each GPIO pin has two corresponding bits & PA5 responds to bit 10 & 11.

ODR - Output Data Register.

```
// Turn on LED
GPIOA -> ODR |= (1<<5);
// Sets the 5th bit, turning the LED on.

// Turn off LED
```

```
GPIOA -> ODR &= ~(1 << 5);
// Clears the 5th bit, turning the LED off
```

GPIO as Input :

```
// Enable GPIOA clock
RCC -> AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
```

Same as output example.

```
// Configure PA6 as input
GPIOA -> MODER &= ~(0b11 << (6*2));
// Sets the 6th pin to input mode.
```

0b11 is shifted left by 12 bits and then negated. This value is ANDed with current MODER register value.

&= : bitwise AND operator. performs bitwise AND between the current value of GPIOA -> MODER and the value on the right side of operator.

0b11 = 3

<< 6*2 = left shift

0b11 << (6*2) : Shifting 12 bits.

<< (6 * 2): This is a left shift operator. It shifts the bits of the number on the left (0b11, or three) to the left by the number of positions specified on the right. In this case, it's shifting the bits 12 positions to the left (because 6 * 2 = 12). This is done because the MODER register uses two bits for each pin, so pin 6 would be at bit positions 12 (for the lower bit) and 13 (for the upper bit).

~: This is a bitwise NOT operator. It flips all the bits of the number it's applied to. Here, it's flipping the bits of the value we've created with `` `0b11 << (6 * 2)` ``. The effect is that we have a number where all bits are set to 1, except for the two bits that correspond to pin 6, which are set to 0.

USART/UART

Thursday, 18 May, 2023 10:51 AM

USART – Universal Synchronous Asynchronous Receiver Transmitter.

Allows serial communication between microcontroller and other devices.

sending data one bit at a time.

In asynchronous mode data is transferred without a clock signal.

In synchronous transmitter & receiver both share a common clock signal to synchronize the transfer process.

USART → Full duplex
→ Half duplex
→ simplex

USART Configuration steps:

1. Enable USART Clock
2. Enable GPIO Pins

③ Enable USART Parameters

④ Enable USART

→ setting appropriate bit in RCC.

→ USART needs two GPIO pins

→ TX (Transmit)

→ RX (Receive)

→ baud rate

→ word length

→ start bit, stop bit

→ panity bit

→ setting UE bit in USART_CR1

USART Transmission Steps:

→ Check TXE Flag

→ Transmit data register empty.

→ Write Data

→ Check TC flag.

→ Transmission Complete

→ Write the data to be transmitted to USART_DR

→ DR: data Register

wait for the flag to be set in the USART_SR register

USART Reception steps:

- Check RXNE Flag
- Read data

RXNE = Read data Register not Empty.

Transmitting a character 'A' using USART2:

```
// Enable USART2 clock

RCC -> APB1ENR |=
            RCC_APB1ENR_USART2EN;
```

- Basic enabling. Remember GPIO ב AHB, USART ב APB

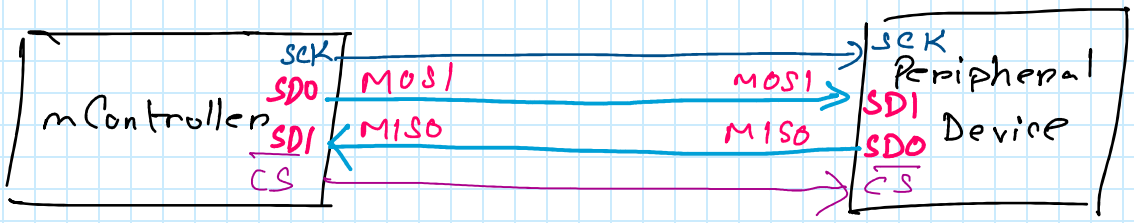
```
// Configure GPIO pins for USART2 TX/Rx
// Assume PA2 for TX and PA3 for RX
GPIOA -> MODER |= (0b10 << 4);
// Set PA2 to alternate function
// mode.
```

Alternate function allows the pin to be used for a peripheral function (USART2 TX in this case). The 0b10 is shifted left by 4 bits because the MODER register has 2 bits for each GPIO pin, PA2 corresponds to pin 4 & 5.

SPI - Serial Peripheral Interface

Friday, 19 May, 2023 11:42 AM

Controller ไม่ก็ master ส่วน device ก็ slave.



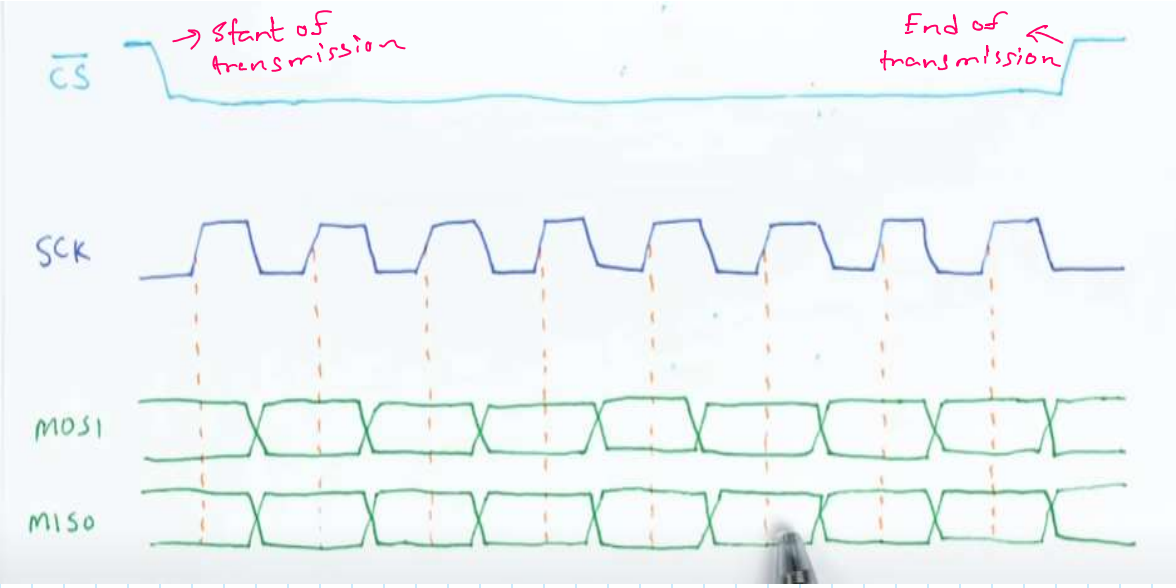
clock signal always generated by the controller.

$\overline{CS}$ = Chip Select

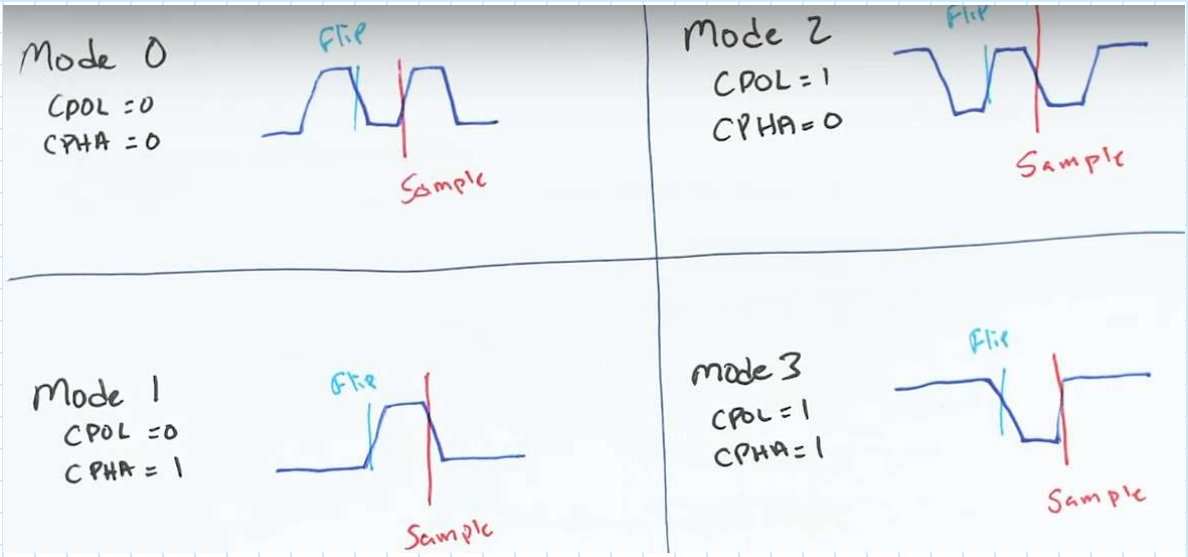
The SPI interface use the following pins:

CSB = Chip Select Bar ($\overline{CS}$). active low
SDI = Serial Data Input
SDO = Serial Data Output
SCK = Serial Clock

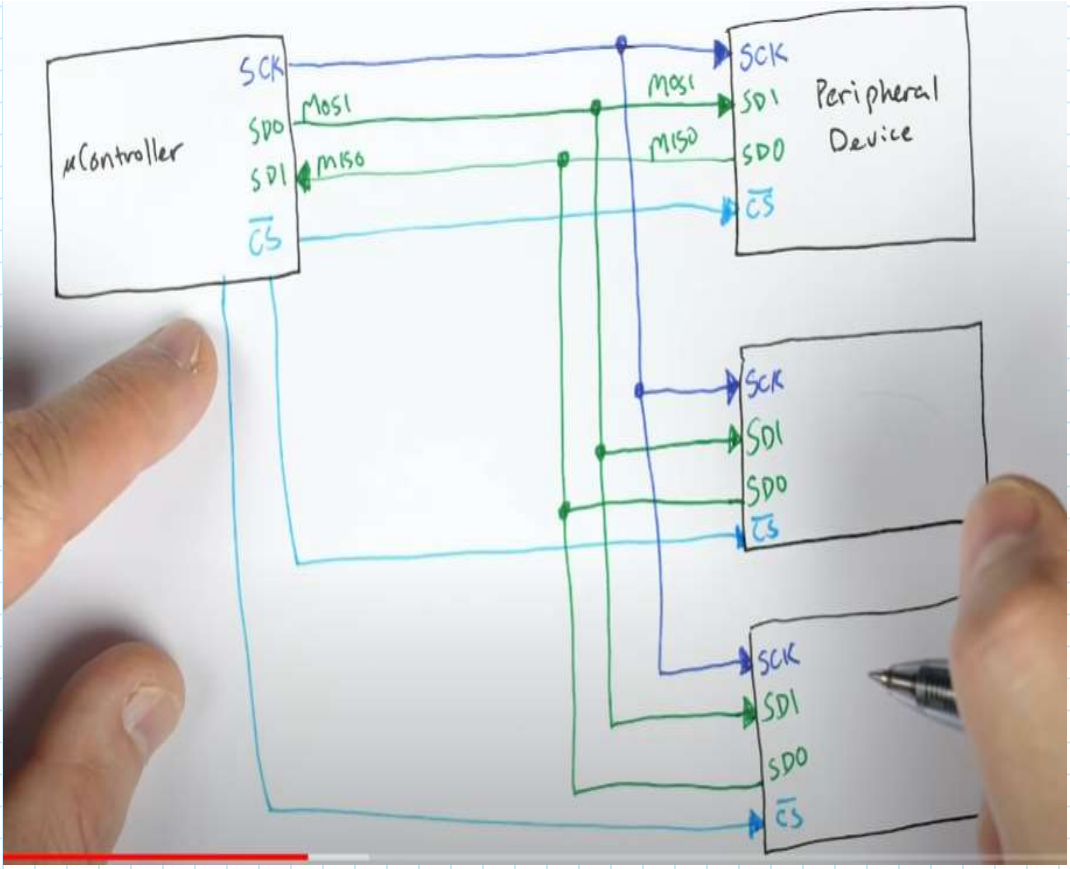
MOSI = Master Out Slave In
MISO = " In " Out



There is no specific rule on how the clock is actually interpreted.



We can have multiple devices having the same bus system connated together. Using chip select ($\overline{CS}$) we can configure কোন device এর সাথে communication করবো।



Enabling the SPI Clock:

```
RCC -> APB2ENR |= RCC_APB2ENR_
                  SPI1EN;
```

Configure GPIO pins for SPI:

Suppose our SPI1 ~~Pins~~ uses the following pins.

SCK: PA5
MISO: PA6
MOSI: PA7

We need to set these pins to alternate function mode & set

We need to set these pins to alternate function mode & set their alternate function to SPI1.

```
// Enable the GPIOA clock
RCC->AHB1ENR |= RCC_AHB1ENR_
                GPIOAEN;

// Set PA5, PA6 and PA7 to
// alternate function mode.
GPIOA->MODER |= (0b10 << (5*2)) |
                (0b10 << (6*2)) |
                (0b10 << (7*2));
```

Most of the GPIO pins in the STM32 microcontrollers can serve multiple functions besides just being a simple input or output pin. They can be used for a variety of other functions, such as serving as an interface for communication protocols like USART, SPI, I2C, or even connecting to the system clock. The specific function of a pin is selected through its alternate function configuration.

The configuration for alternate functions is handled by the GPIOx_AFRL and GPIOx_AFRH registers, where 'x' is the port letter. These registers define the mapping of the I/O pins with the alternate function. The AFRL register is used for pins 0 to 7 and the AFRH register is used for pins 8 to 15.

```
// Set the alternate functions of PA5, PA6, and PA7 to SPI1
GPIOA->AFR[0] |= (0b0101 << (5 * 4)) | (0b0101 << (6 * 4)) | (0b0101 << (7 *
```

Let's break it down:

- **`GPIOA->AFR[0]`**: We are selecting the GPIO port A and its Alternate function low register (AFRL) because we're dealing with pins under 8 (PA5, PA6, PA7).
- **`(0b0101 << (5 * 4))`**: We are shifting the binary value **`0101`** to the left by 20 bits. This binary number **`0101`** corresponds to AF5 in the datasheet, which for PA5 is SPI1_SCK.
- **`| (0b0101 << (6 * 4))`**: The same thing is happening for PA6. We shift **`0101`** (representing AF5, which maps to SPI1_MISO) to the left by 24 bits.
- **`| (0b0101 << (7 * 4))`**: Likewise for PA7, we shift **`0101`** (representing AF5, which maps to SPI1_MOSI) to the left by 28 bits.

The `|=` operator sets these values in the AFR[0] register without altering the bits meant for other pins.

So in essence, this line of code sets PA5, PA6, and PA7 to function as SPI1_SCK, SPI1_MISO, and SPI1_MOSI respectively.

3. **Configure the SPI Peripheral**

Next, we need to configure the SPI peripheral. For simplicity, let's start with the following settings:

- Master mode
- SCK idle state low (CPOL = 0)
- Capture data on the first edge (CPHA = 0)
- Data frame format: 8 bits
- SPI enable

c Copy code

```c
// Set the SPI to master mode, set the baud rate prescaler to 256,
// set the CPOL and CPHA to 0, and set the data frame format to 8 bits
SPI1->CR1 = SPI_CR1_MSTR | SPI_CR1_BR_2 | SPI_CR1_BR_1 | SPI_CR1_BR_0;

// Enable the SPI
SPI1->CR1 |= SPI_CR1_SPE;
```

How to set or change prescalar in general? What is prescalar baud rate, what is cpol an cpha.

Prescalers are used in microcontrollers to divide the frequency of the clock signal. This can be very useful to slow down the clock for certain peripherals or operations to match their operating speeds. A prescaler can be seen as a counter that increments each time the system clock ticks. When this counter reaches a predetermined value, the counter overflows and the connected peripheral or timer registers a tick.

In STM32 microcontrollers, prescalers are commonly used with timers and peripheral interfaces like SPI, I2C, and USART. To set or change a prescaler, you typically write to the appropriate register.

For example, in the case of a timer, you might have a register such as TIMx_PSC (Timer x Prescaler Register) which you can set with a prescaler value.

c Copy code

```
TIM2->PSC = 15999;  // set prescaler for TIM2 to 16000 (0-based), 1 tick per
```

In case of peripherals like SPI, USART, the prescaler or baud rate is set through a particular register.

c Copy code

```
SPI1->CR1 |= SPI_CR1_BR_0 | SPI_CR1_BR_1; // SPI1 baud rate fPCLK/64
```

Baud Rate: In terms of serial communication (like USART), baud rate is the speed of data transmission measured in symbols per second or pulses per second. It is set using a prescaler to match the speed of the receiving end for proper data transfer.

CPOL (Clock Polarity) and CPHA (Clock Phase): CPOL and CPHA are configurations for synchronous serial communication protocols like SPI.

CPOL determines the base value of the clock when it's idle (not transmitting data).

- If CPOL=0, the clock is at 0 when idle (also known as active high configuration).
- If CPOL=1, the clock is at 1 when idle (also known as active low configuration).

CPHA determines whether data is sampled at the first edge or second edge of the clock signal.

- If CPHA=0, data is sampled at the first clock transition (also known as the leading edge).
- If CPHA=1, data is sampled at the second clock transition (also known as the trailing edge).

These configurations affect how devices using synchronous serial communication are connected and exchange data. The devices need to agree on these settings to successfully communicate.

4. Communicating with a Slave Device

Finally, we can send and receive data.

c

```c
// Pull the CS line low to select the slave
GPIOA->ODR &= ~(1 << 4);

// Wait until the transmit buffer is empty
while (!(SPI1->SR & SPI_SR_TXE));

// Send the data
SPI1->DR = 0x55;

// Wait until a reply has been received
while (!(SPI1->SR & SPI_SR_RXNE));

// Read the received data
uint8_t receivedData = SPI1->DR;

// Pull the CS line high to deselect the slave
GPIOA->ODR |= (1 << 4);
```

This code snippet is implementing a basic SPI data transmission in master mode on an STM32 microcontroller. Let's break it down line by line:

1. `GPIOA->ODR &= ~(1 << 4);`
   This line is pulling the Chip Select (CS, also called Slave Select, SS) line low to select the slave device. The `ODR` register (Output Data Register) is used to set the state of the GPIO pins (PA4 in this case). The operation `&= ~(1 << 4)` is setting the 4th bit of the ODR register to 0, which means PA4 is driven low.
2. `while (!(SPI1->SR & SPI_SR_TXE));`
   This line is waiting until the Transmit Buffer Empty flag (TXE) is set. The TXE flag in the Status Register (SR) indicates that the transmit data register is empty and ready for the next data.
3. `SPI1->DR = 0x55;`
   This line is sending the data. The `DR` register (Data Register) is used for data transmission. The data to be transmitted is written to this register (0x55 in this case).
4. `while (!(SPI1->SR & SPI_SR_RXNE));`
   This line is waiting until a reply has been received. The Receive Buffer Not Empty flag (RXNE) in the SR register indicates that received data is ready to be read.
5. `uint8_t receivedData = SPI1->DR;`
   This line is reading the received data. After a successful reception, the received data can be read from the DR register.
6. `GPIOA->ODR |= (1 << 4);`
   This line is pulling the CS line high to deselect the slave. The operation `|= (1 << 4)` is setting the 4th bit of the ODR register to 1, which means PA4 is driven high.

This example demonstrates a single data exchange (send one byte, receive one byte) over SPI. A real-world application might involve more complex transactions, such as sending/receiving multiple bytes, handling different types of data, or managing multiple slave devices.

Reference Manual?

SPI Supports → Simplex
→ Half Duplex
→ Full Duplex

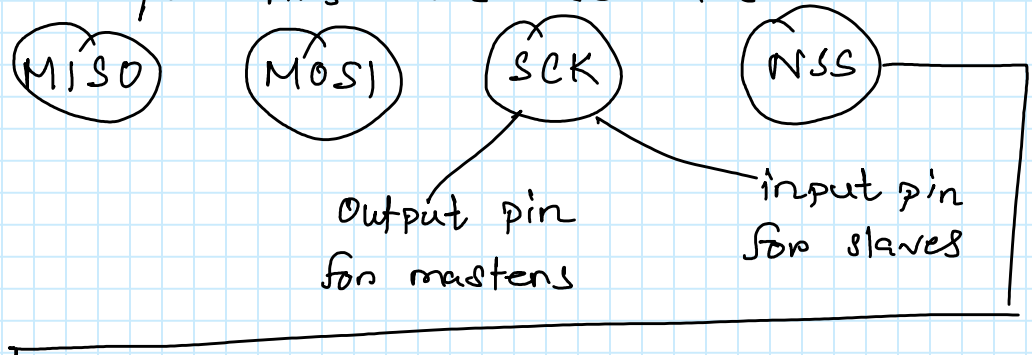
Capable of multi master

Full Duplex syn transfer on 3 lines
Half " " " " 2 lines

Max Slave mode freq ≤ $f_{PCLK}/2$

Max Slave mode freq = $f_{PCLK}/2$

Four I/O Pins are dedicated to SPI comm.



↳ Slave Select Pin
↳ Select individual slave device.
↳ detect conflict between masters.
↳ Depends on SPI and NSS settings

Schematic Diagram

System Design at first

Methodoly

Publication target:

① Scientific explanation
↳ Prove mathematically
↳ conner cases

IEEE / Sciencedirect

work & keep documentation

Schematic diagram

↓

system design

↓

methodologies

Contribution → outcome

Intro ও state of the art আলোচনা, existing system এর problems গুলো বলা।

Methodology → tasks objective

Detailed design

Block diagram

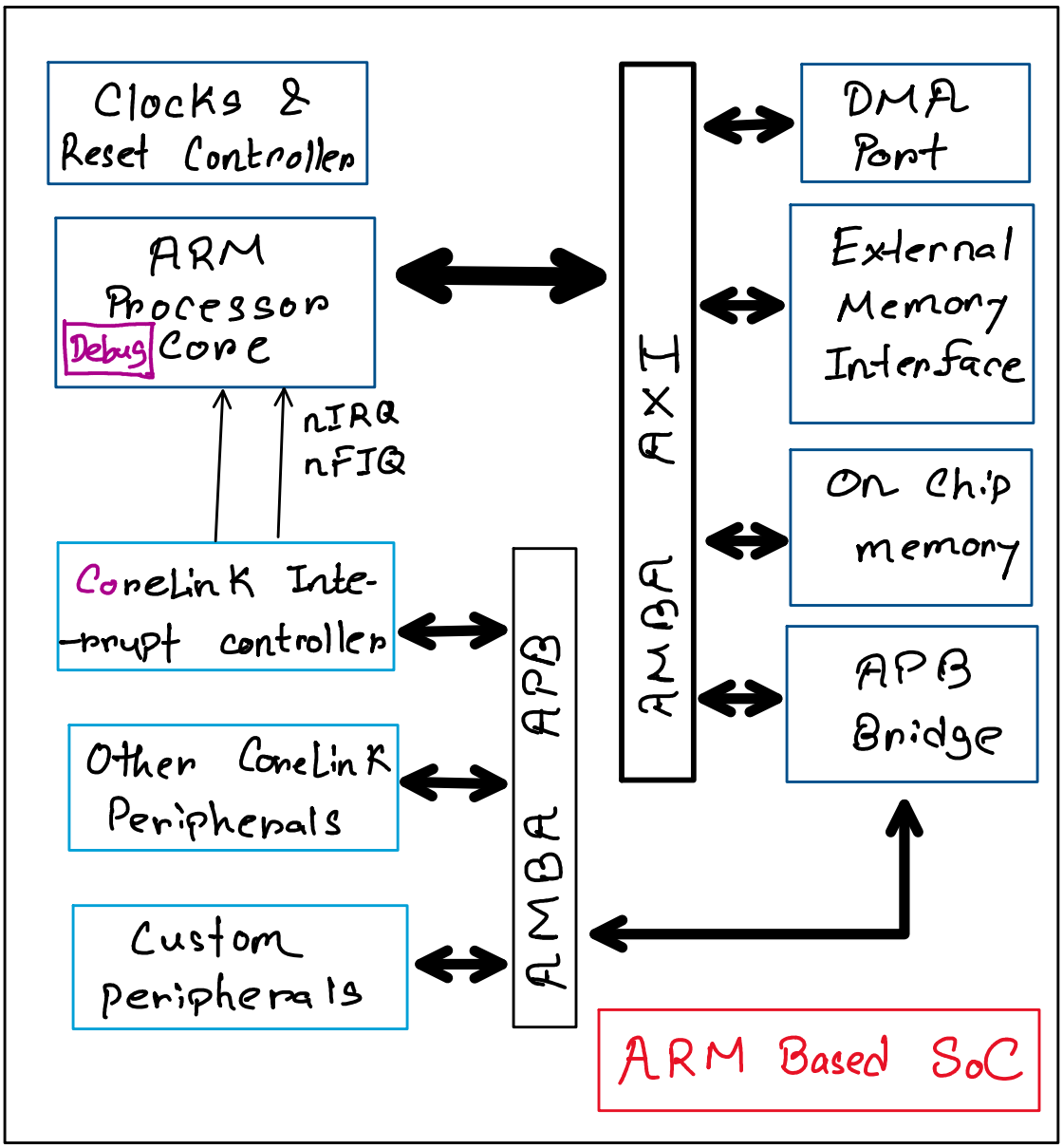
The ARM

Wednesday, 8 February, 2023 8:01 PM

Inside an ARM based system:

SoC = System on a Chip

ARM based SoC means device with an ARM processor



→ Core is generally deeply embedded in the device.

→ Debug port is often the only exposed connection.

→ ARM has interrupt inputs.

↳ An interrupt I/O is a process of data transfer in which an external device or a peripheral informs the CPU that it is ready for communication and requests the attention of CPU.

→ Within the device the components are connected together using an on chip interconnected bus architecture.

→ AMBA specifies two buses.

↳ Advanced Microcontroller Bus Architecture.

→ AXI is high performance system bus.

→ APB is low power peripheral bus.

↳ used to collect all the peripherals.

⇒ AXI for memories & other high speed devices.

Development of the ARM architecture:

[v4T → v5TE → v6 → v7]

v7 → Thumb-2
NEON
TrustZone
Virtualization

But hear me out. ARM processor is used in small devices. microcontrollers & then in high end devices like phones & laptops.

Therefore we decided to introduce Architecture profiles.

Architecture profiles:

(i) v7-A (Applications) : NEON

→ provides features that can support an operating system

→ Build for application processes

(ii) v7-R (Real time): Hardware divide

→ High performance predictable real time

(iii) v7-M (Microcontroller)

→ Cost sensitive microcontroller applications.

Now you know why we use CORTEX M-4.

ARM v7M Profile

→ Born to serve microcontroller.

→ Lowest gate count entry point
→ Deterministic & predictable behaviour.
→ Deeply embedded use

Data sizes & instruction sets:

⇒ ARM is a RISC architecture

- Most instructions execute in a single cycle
- Orthogonal register set
  - Instruction type & addressing mode vary independently
- Load store architecture
  - instruction set architecture that divides instructions in two categories
    - memory access
      - load & store between memory & registers
    - ALU operations
      - only occur between registers

⇒ ARM is a 32 bit load store architecture

The only memory accesses are loads and stores.
Most internal registers are 32 bit wide.

⇒ In ARM "word" means 32 bits.

⇒ Most ARM cores implement two instruction systems.

- 32 bit ARM Instruction set
- 16/32 " Thumb " "

Processor Modes:

⇒ Most ARM cores have 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 privileged mode.

07EBB948-348C-460...

Processor Modes

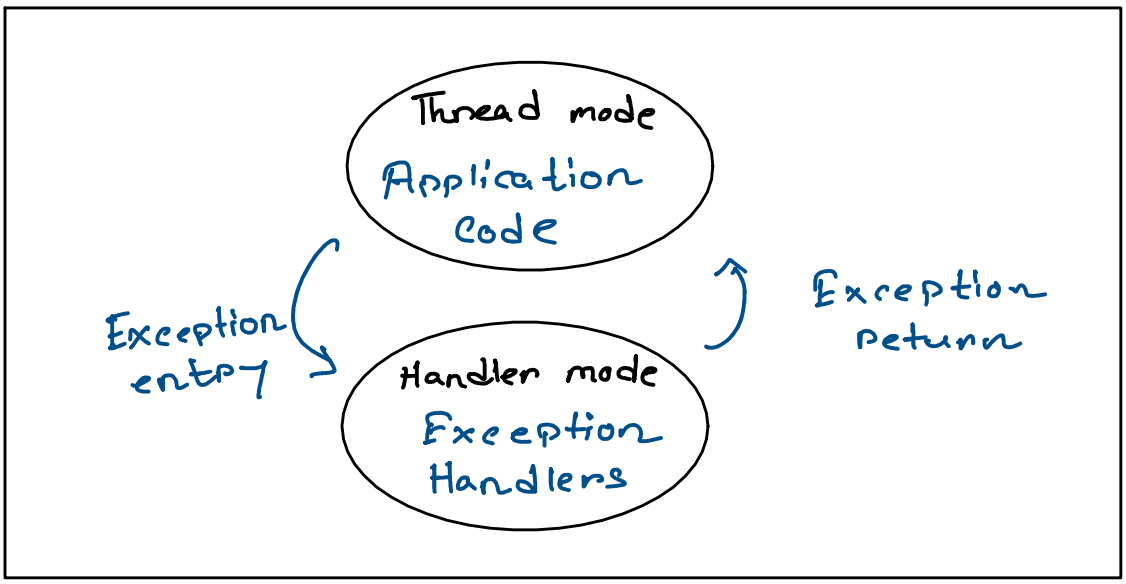
- Most ARM cores have seven basic operating modes
  - Each mode has access to its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Mode	Description
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed
FIQ	Entered when a high priority (fast) interrupt is raised
IRQ	Entered when a normal priority 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

Privileged modes

Unprivileged mode

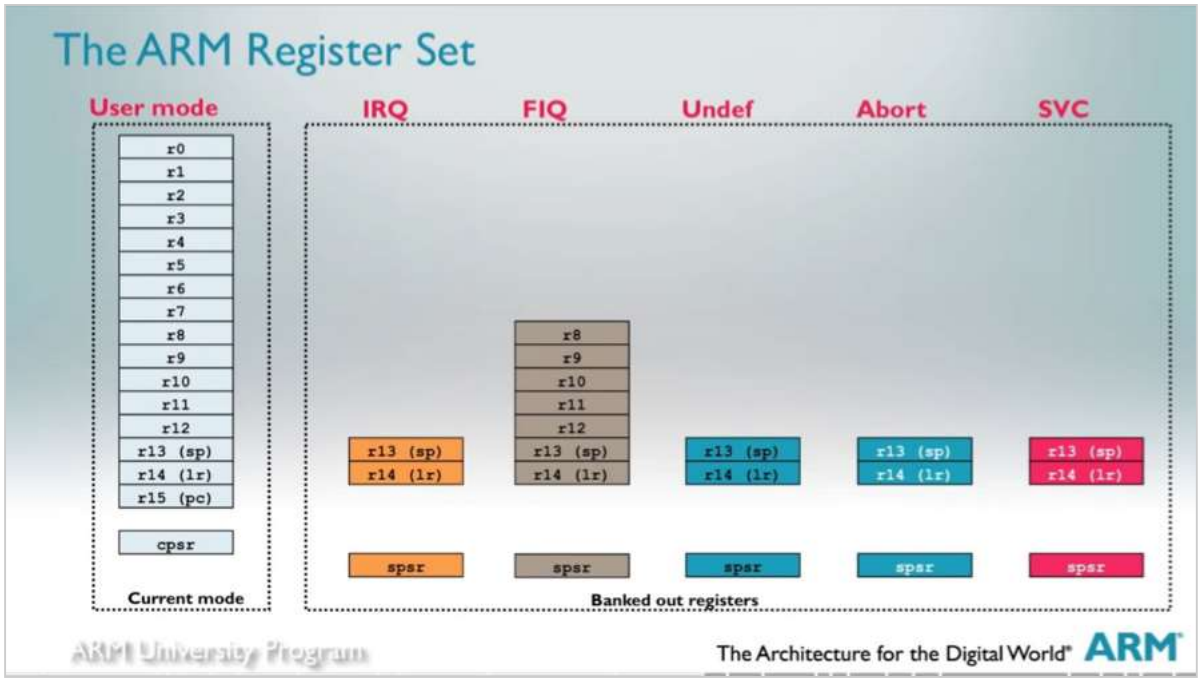
Arm v7 microcontroller profile defines just two modes.



Thread mode is unprivileged.

36B64DF4-633E-424...

How modes & registers interact:



IRQ - Interrupt Request

FIQ - Fast " "

↳ reserved for single, high priority interrupt source that requires a guaranteed fast response time.

IRQ - used for all the others interrupt in the system.

IRQ mode has its own locations & registers to store its relevant information. ~~the~~ its own private stack.

SPSR - Saved Program Status Register

↳ holds a snapshot of the system at the moment when the

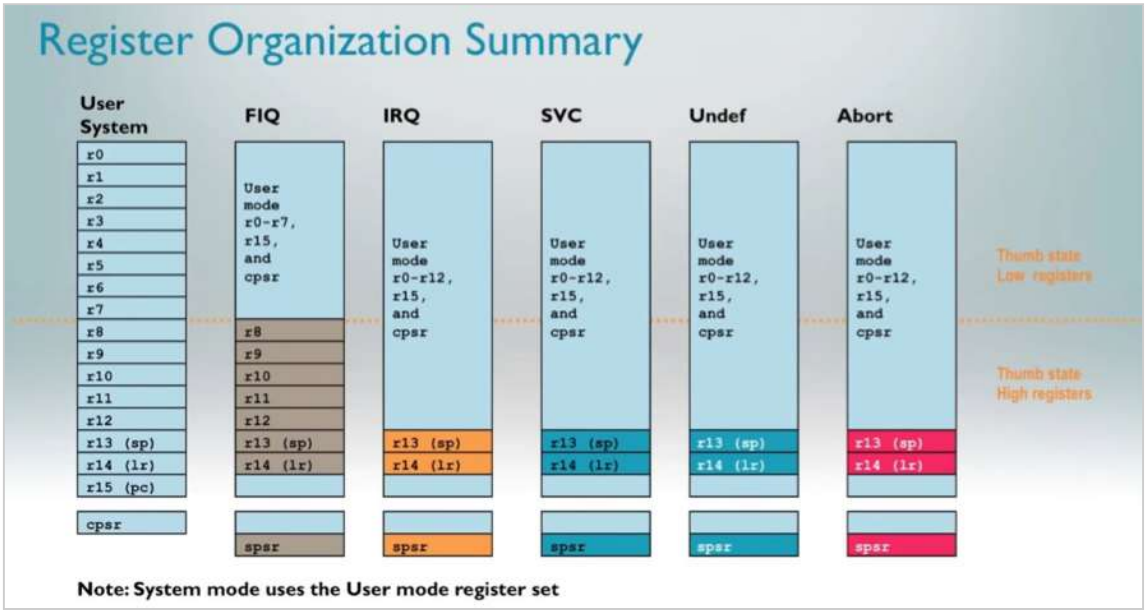
at the moment when the exception is taking place. This makes returning to where we were very easy.

We just have to switch the registers back. Restore CPSR. And restore the program counter. From the values that we have saved.

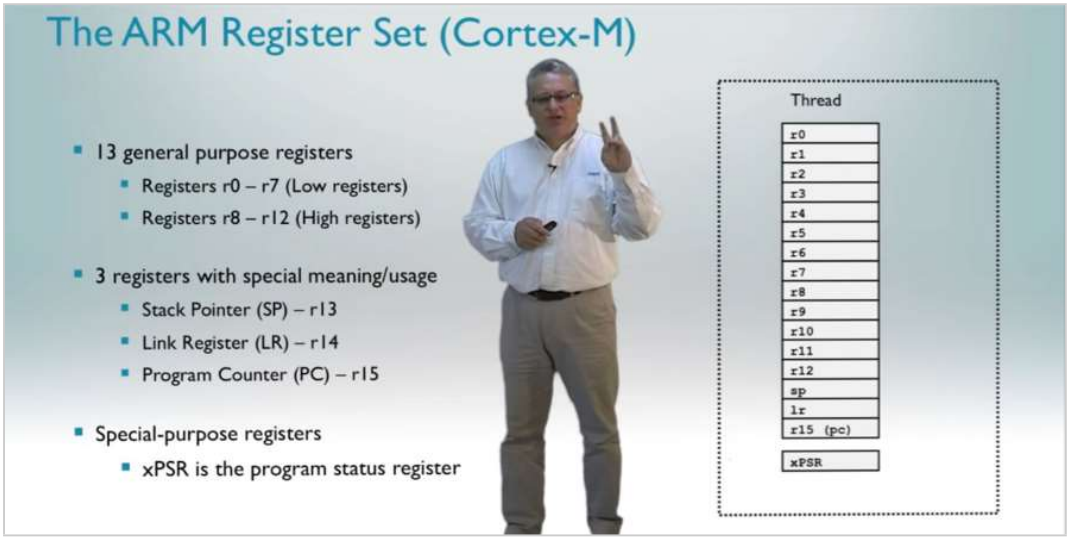
FIQ - r8 → r12 are additional private registers for FIQ mode.

System mode uses the same registers as the user mode. System is privileged. user is not.

13BE3C92-8D41-4E8...

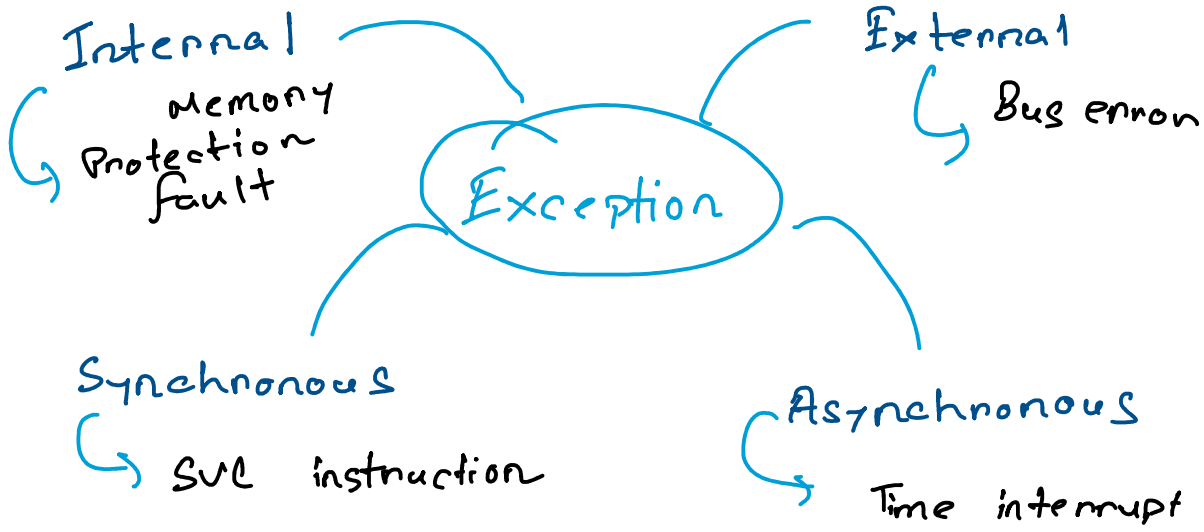


6CF1B3B2-EF16-4D3...



Exception Handling:

Exception causes interrupt in normal program flow.



Assembly Programming

Thursday, 2 March, 2023 6:44 PM

Data Processing Instruction Set:

Arithmetic & Logical operations on data.

ADD:

① Add two registers & store in another

```
ADD  R0, R1, R2;
     ↓
destination register
```

② Add a constant value to a register & store it to another

```
ADD  R0, R1, #5
          ↓
R1 এর সাথে যোগ হয়ে ফলাফল
5 যোগ হবে।
```

③ ADD a register to a memory location and store the result in another register.

```
LDR  R1, [R2]
ADD  R0, R1, R3
```

```
LDR   R1, [R2]
ADD   R0, R1, R3
```

↳ Loads the content of the memory location pointed to by R2 into R1.

↳ Loads a 32 bit word from memory and stores it in register R1.

↳ The address of the word to be loaded is specified indirectly through the contents of the register R2.

For example if 'R2' contains the value 0x1000, the instruction 'LDR R1, [R2]' would load the 32 bit word located at memory address 0x1000 into register R1.

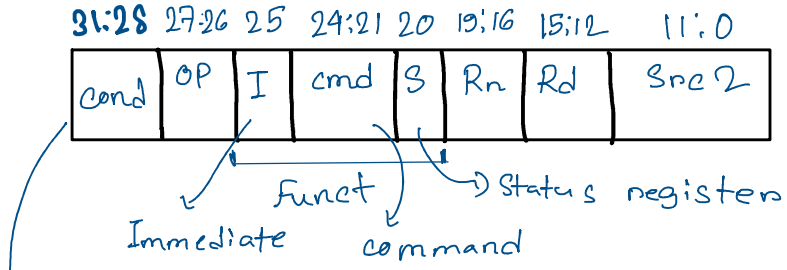
④ Add the contents of 2 memory locations and store the result in a register.

```
LDR   R1, [R2]
LDR   R3, [R4]
ADD   R0, R1, R3
```

ADDEQ R1, R2, R3

↳ Addition takes places based on a condition of flag set previously.
If 0 flag is set R2 is added with R3 and stored in R1.

ADDNE
↳ when zero flag is cleared or not set.



এখানে general opcode for any instruction is (1110) → Remember that.

↳ 1110 00 0

"#" use করে যেখানে value সেখানে আমরা I = 1
↳ immediate

cmd → specifies ARM data processing instructions like AND OR SUB CMP MOV MVN

sh field encodings

Instruction	sh	Operation
LSL	00	Logical Shift Left
LSR	01	" " Right
ASR	10	Arith. " "
RoR	11	Rotate right.