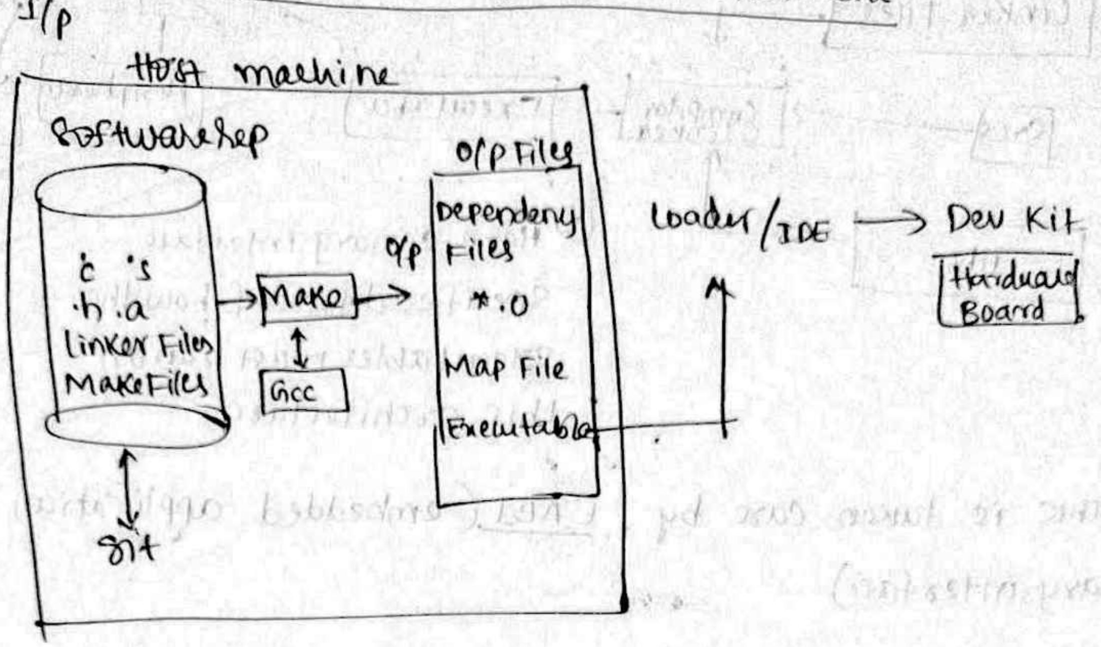
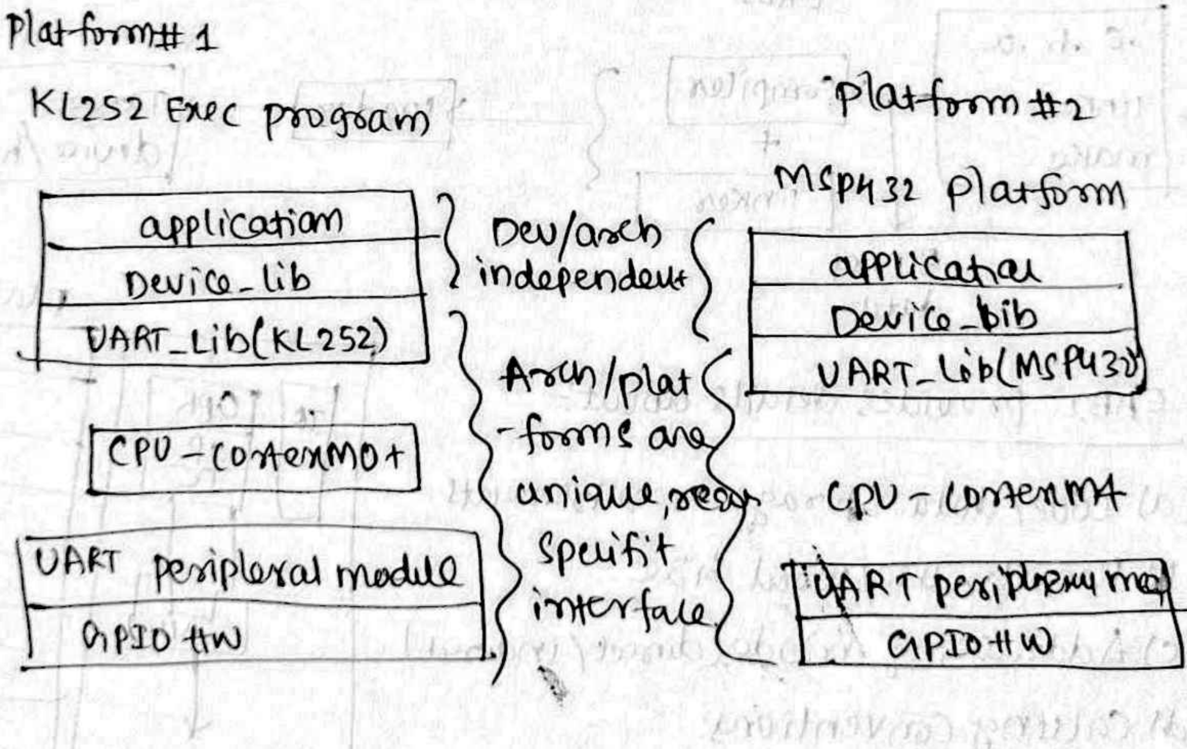
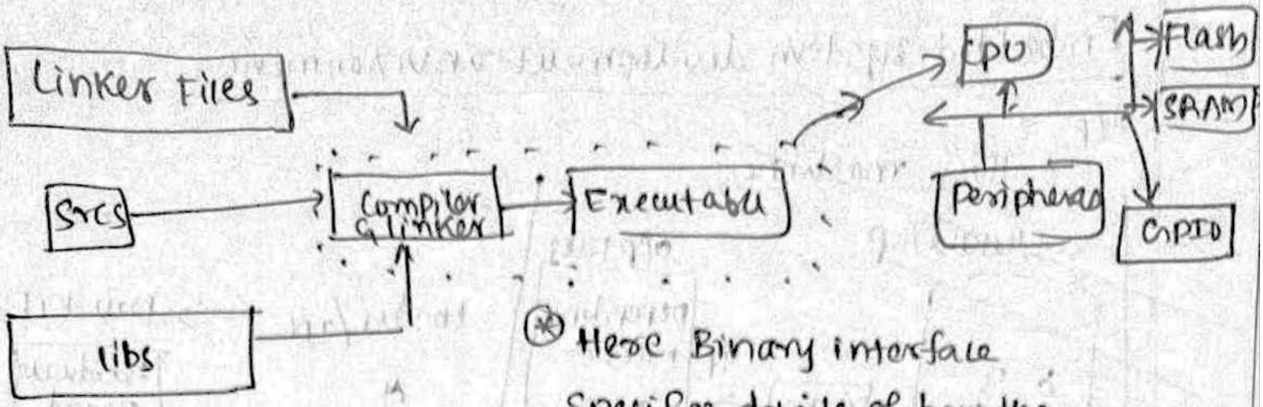
Embedded system development environment



→ Target is to write as much software code as platform as well as architecture independent

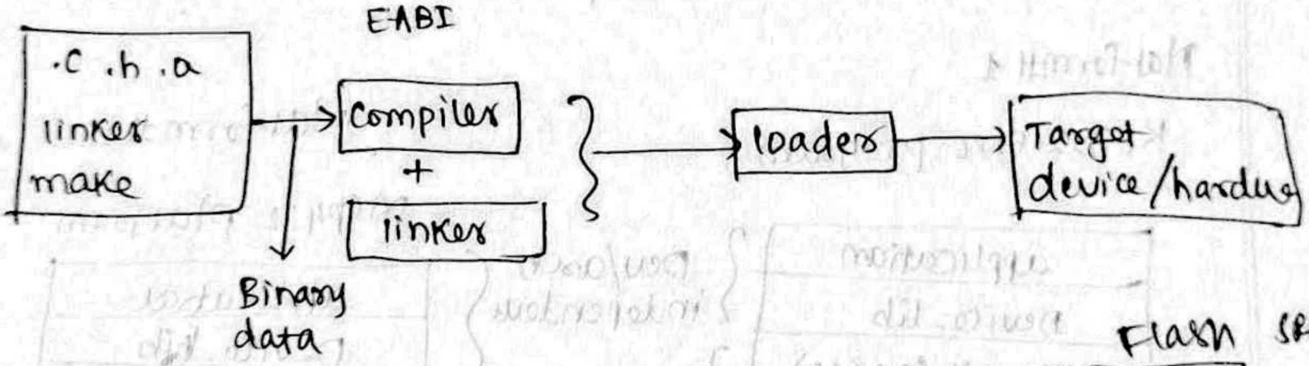




⊛ Here, Binary interface Specifies details of how the executables must run on this architecture

⊛ This is taken care by <u>EABI</u> (embedded application binary interface)

⊛ It takes care of compiler to generate the platform based code by taking the binary as input



EABI provides details about:-

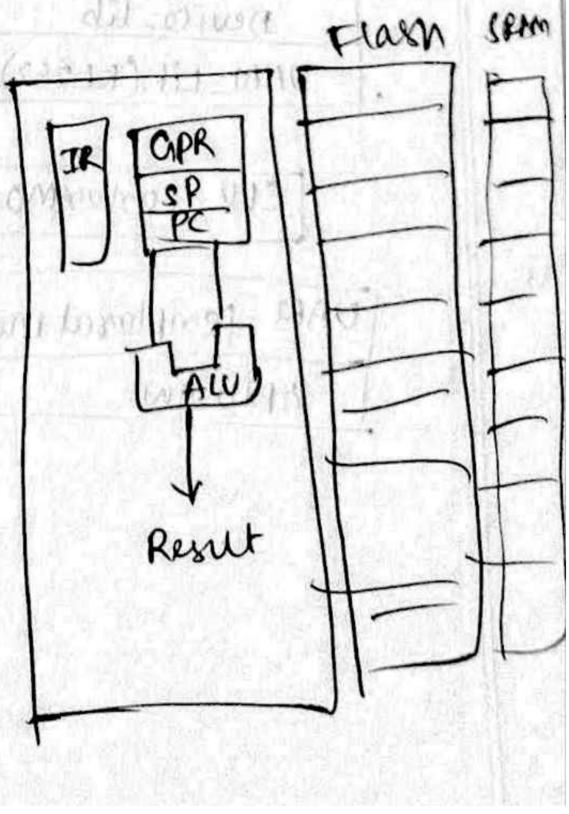
(a) Code/data storage requirements

(b) Register use/word size

(c) Addressing mode (direct/indirect)

(d) Calling conventions

(e) Helper Functions and libraries



→ Any architecture is designed to implement ~~execut~~ assembly language

Eg:- CISC (Complex instruction set Computer)

RISC (Reduced instruchan set Computen

ARM → often 32-bit/64-bit architecture

word sizes

Two poplolar tems are 1. Instruction

2. word

* Instruction:- Fundamental unit of work/operation

1. arithmetic
2. logical
3. Program flow control
4. Load/store

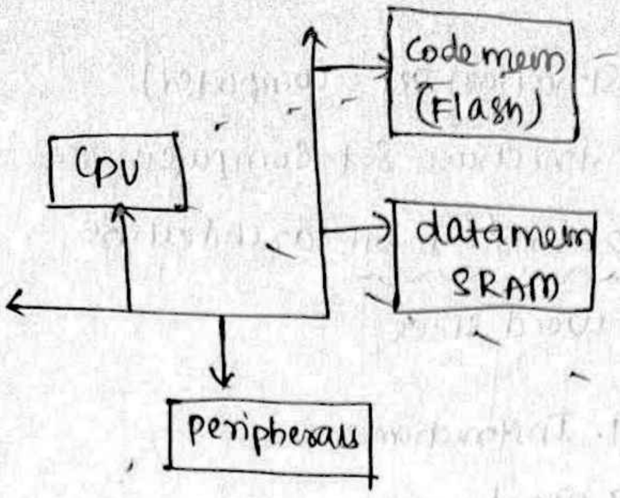
* word: Fundamental operand size for each operation

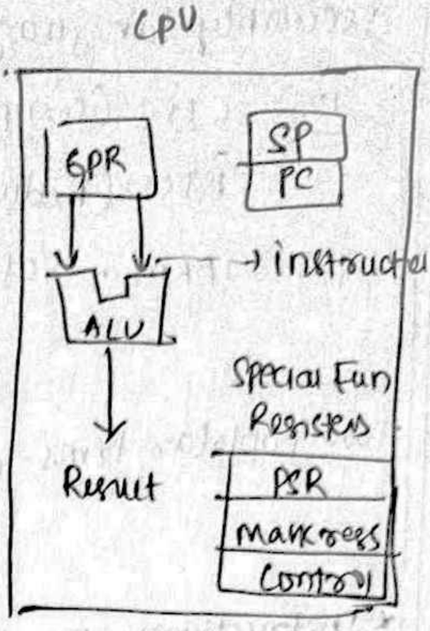
Eg:- If we send "0XFF" as operand then Bypass instruction will be executed

Here word = operand size = 8 bits / 1 byte

and Instruction happened is Bypass instruch

Interacting with memory:-





Different type of memories

(a) code memory (Flash) which stores code and some data (const, init values)

(b) SRAM memory (data memory) stores data like locals, Globals, static which get initialised at the run time.

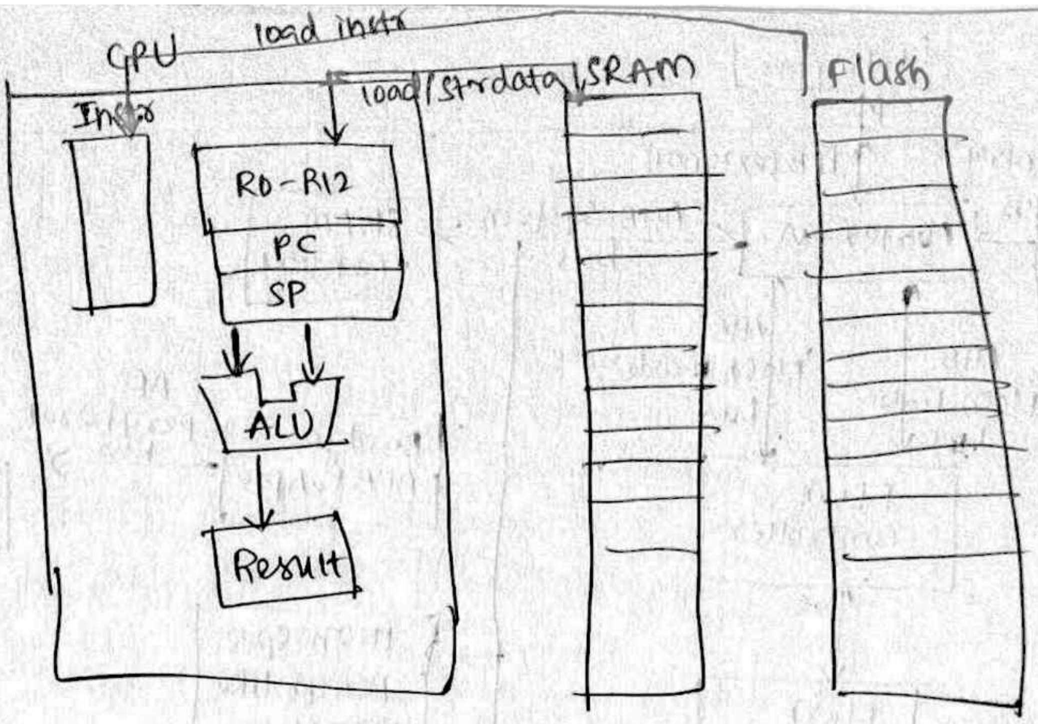
(c) General peripheral registers

(d) CPU core registers like General purpose and special purpose

16 Registers (R0 - R12, R13(SP), R14($\overset{LR}{\cancel{PC}}$), R15(PC))

5 special purpose registers

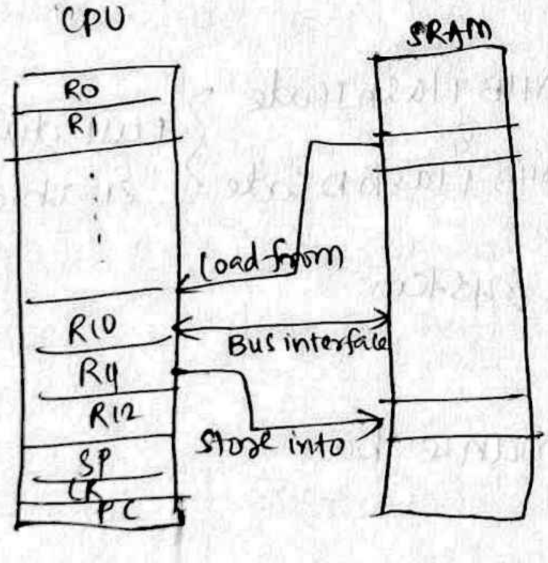
⊛ For the cpo operations like arithmentic, algos etc.., these registers are not enough hence it takes help from Flash and SRAM.



→ CPU uses load to load data/instr from memory and store to store the data back to memory

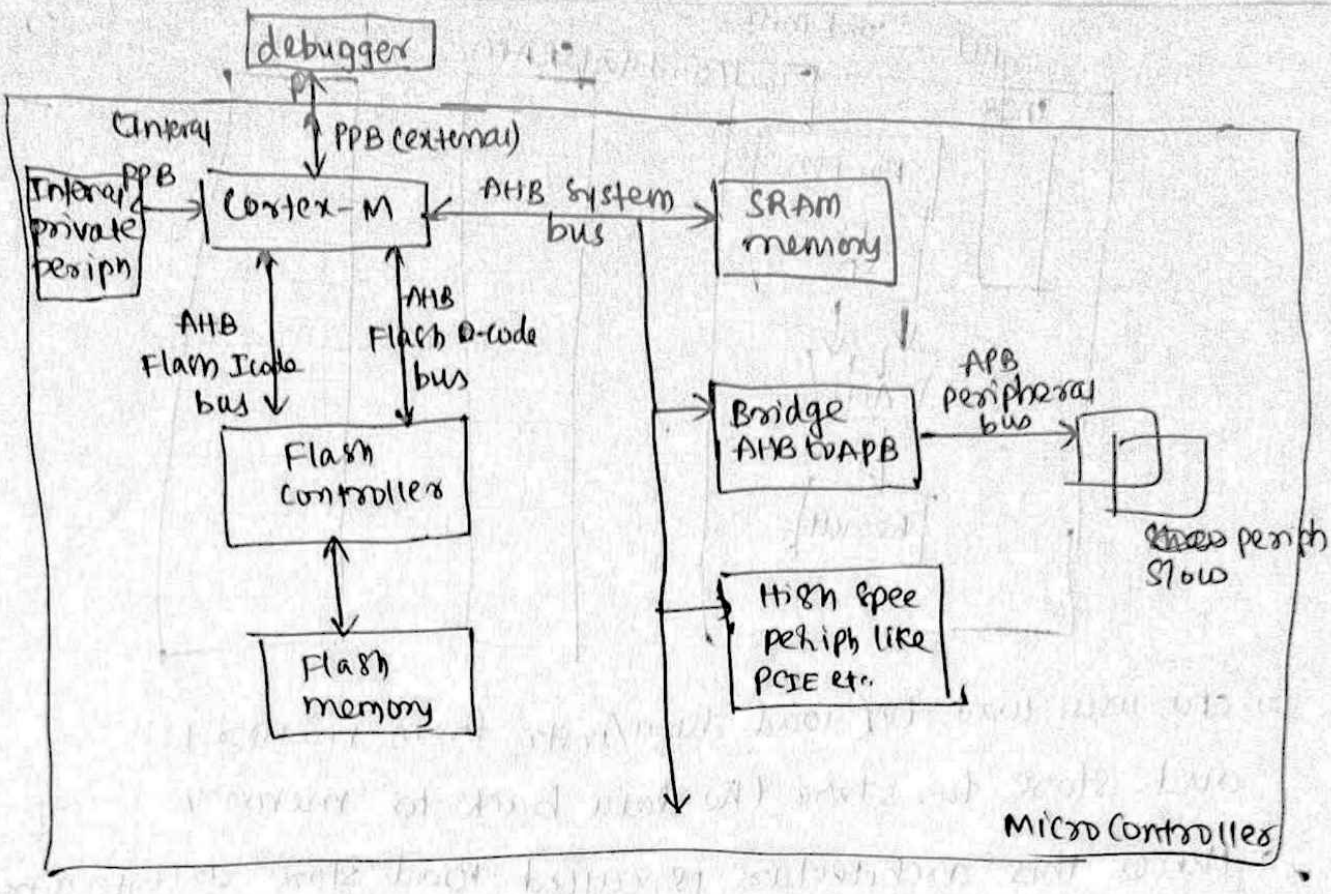
→ Hence this architecture is called load-store architecture

Read-modify-write



→ any manipulations lesser than byte is handled by hardware

→ NO CPU load/store



Inside cortex M processor, interconnection follow ~~AB~~ AMBA protocol, AMBA → advanced microcontroller bus architecture

From cortex M processor,

To code mem (Flash) → AHB Flash Icode & AHB Flash Dcode } Full duplex enables

To SRAM (data) → AHB system bus

AHB ⇒ AMBA high performance bus

APB ⇒ AMBA periph bus.

~~AB~~ AHB ⇒ $\boxed{\text{Bridge}}$ ⇒ APB.
AHB to APB

also cortex M to External PPB to debugger
to Internal PPB to. Core private periph

Memory alignment:-

⇒ load-Store architecture

→ Load: data is loaded into CPU

→ data is operated on

→ Store: Data is stored back into memory

⊛ Each byte has unique address

⊛ So, in ARM32 architecture $2^{32} \Rightarrow$ 4GB of addresses are possible

⊛ Most of the math applications are performed with 1Byte, half word and word size data

⊛ ISA gives 3 types of load-stores such as byte, half word and word load stores

Assembly load/store instr

LDR → load word

STR → store word

LDRH → load unsigned half word

LDRSH → load signed half word

STRH → store unsigned half word

STRSH → store signed half word

LDRB → load unsigned byte

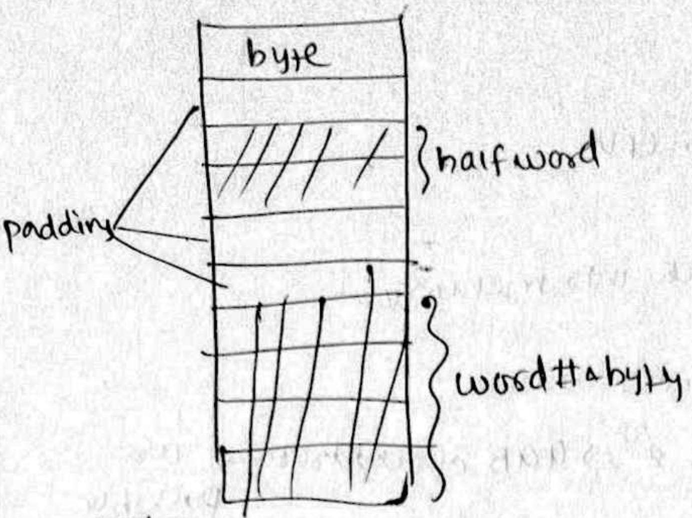
LDRSB → load signed byte

STB → store unsigned byte

STSB → store signed byte

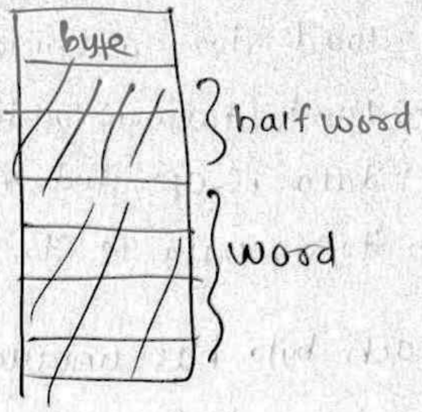
↕ CPU manipulation at one load/ store cycle happens at maximum of <u>word size</u> and min of "<u>byte</u>"

with padding



Data
~~aa~~ alignment
cpu more efficient
memory inefficient

without padding



no alighned but packed
CPU less efficient
memory more efficient

If Speed optimisation → Data alignment
no padding

If memory optimisation → Data packed
→ padding

For speed opti	For memory opti
data alignment	data packed

⊛ ENDIANNESS

Big Endian:- MSB at lowest address
LSB at highest address

Little Endian:- MSB at highest address
LSB at lowes address

Big Endia

0x100	AB
101	CD
102	EF
103	0D

OX ABCDEF0D
↑ MSB ↑ LSB

little Endia

0x100	0D
101	EF
102	CD
103	AB

63020 34441

ARM32

Code mem :- not configurable and it is little endian

data mem :- to be configurable but by default little endian

Compiler attributes

→ attributes can give specific details on how to compile code for
1. Variables
2. Structures and structure variables
3. Functions.

eg
```
struct struct_name {
      ==
      ==
} __attribute__((packed));

int8_t foo __attribute__((aligned(4)));
```

Example with "aligned" keyword:-

```
typedef struct {
        int8_t var1;
        int32_t var2;
        int8_t var3;
    } __attribut__((aligned))
```

⇒ then each variable above takes word size that equals 12 bytes but a $16 = 2^4$ is near to '12'. It will give 16 bytes

If __attribute__((packed))

Then It only occupy 6 bytes

Eg 4:- ~~__attribute__((always_inline))~~ inline

Pragmas

push/ pop:- adds extra optiom to compiler

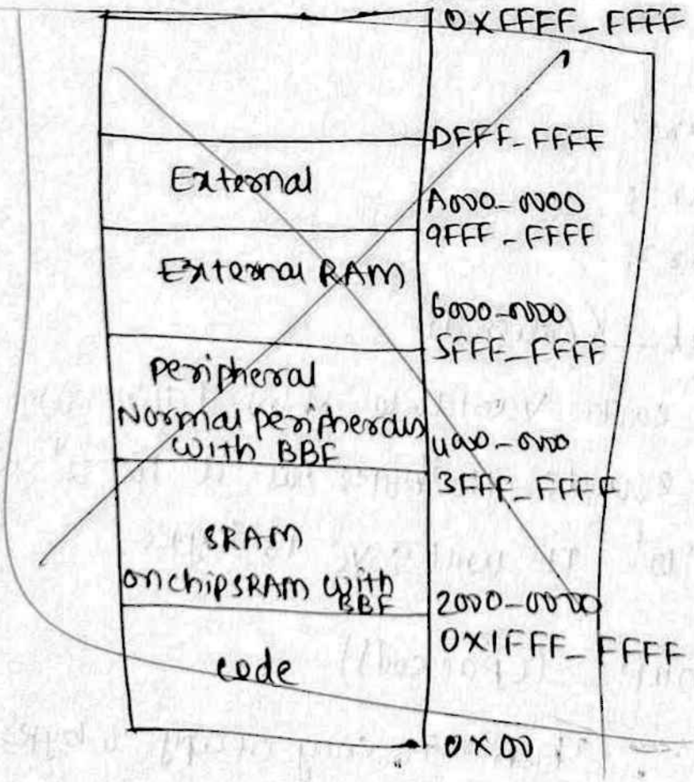
optimise:- Specify a certain level of optimisation block of code

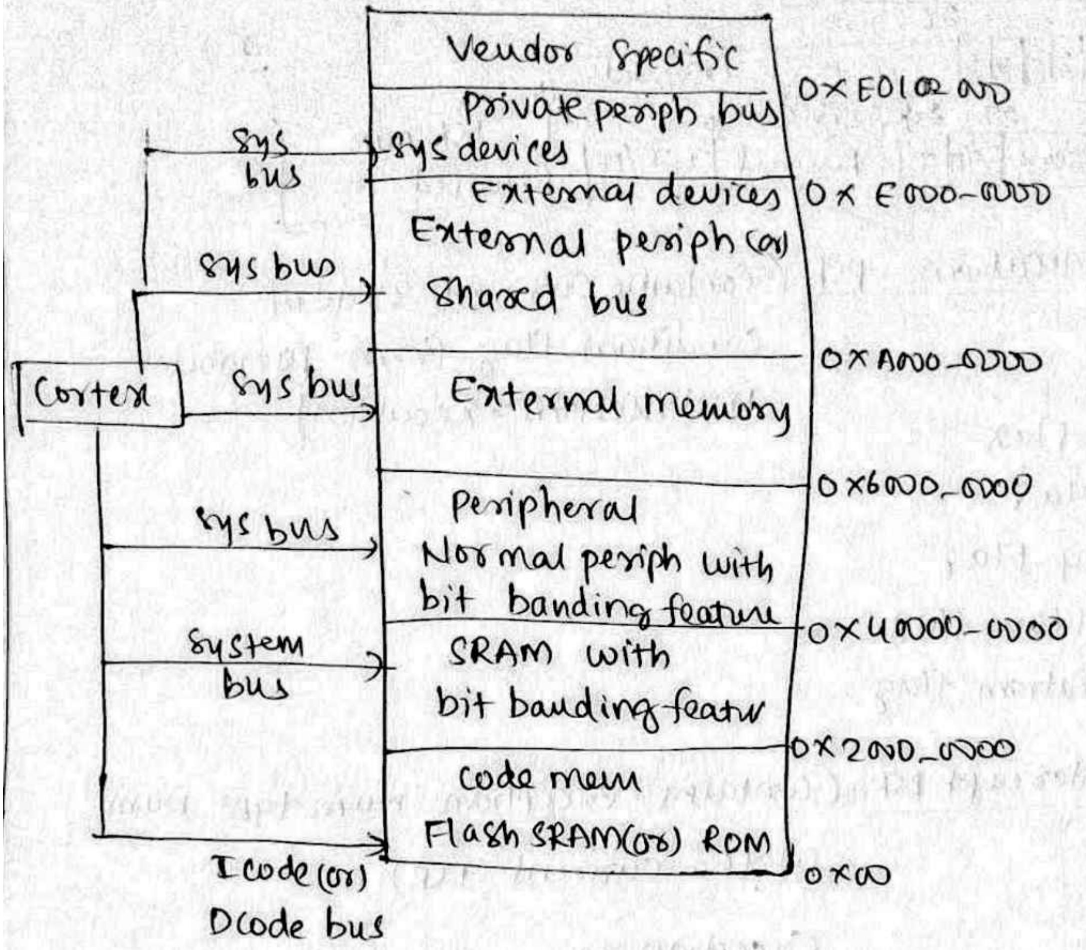
```
#pragma GCC push
#pragma GCC optimise("O0")
 int32        {

       }
 # pragma GCC pop.
```

Memory map and registers

CPU → R0 - R15 ⇒ (R0-R12, R13(SP), R14(LR), R15(PC)) ⇒ general purpose reg

→ PSR, PRIMASK, FAULTMASK, BASEPRI, CONTROl ⇒ Special purpose registers





Typical register memories

→ Internal Core CPU

→ Internal private periph

→ External private periph

→ Gen periph memory

CPU core registers

r0 - r16 ⇒ r0 to r12 GPR
$\underbrace{\quad}$ General purpose reg

r13 → PC

r14 → lr (link reg)

r15 → SP

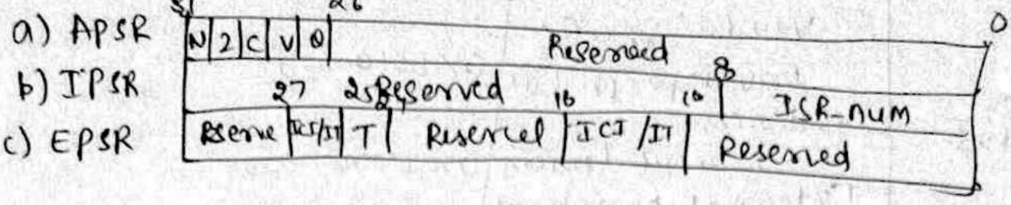
Special reg

PSR → prog Status reg

Exception Mark reg
- → PRIMASK
- → FAULTMASK
- → BASEPRI

Control registers

⊛ program status regr:-



APSR → Application PSR (Contains current state of condition flag from previous instruction execution)

N → neg Flag

Z → zero Flag

C → carry Flag

V → overflow flag

Q → saturation flag

IPSR → interrupt PSR (contains exception ~~num~~ type num of the current ISR)

Exception no

0 = Reset

1 = NMI

2 = Hard fault

3 = mem manag fault

⋮

⊛ Exception mask reg

PRIMASK

Prevent activation of all exceptions with config priority

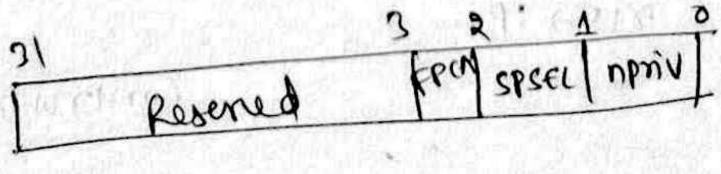
FAULT MASK

prevent activation of all exception with config priority except the NMI

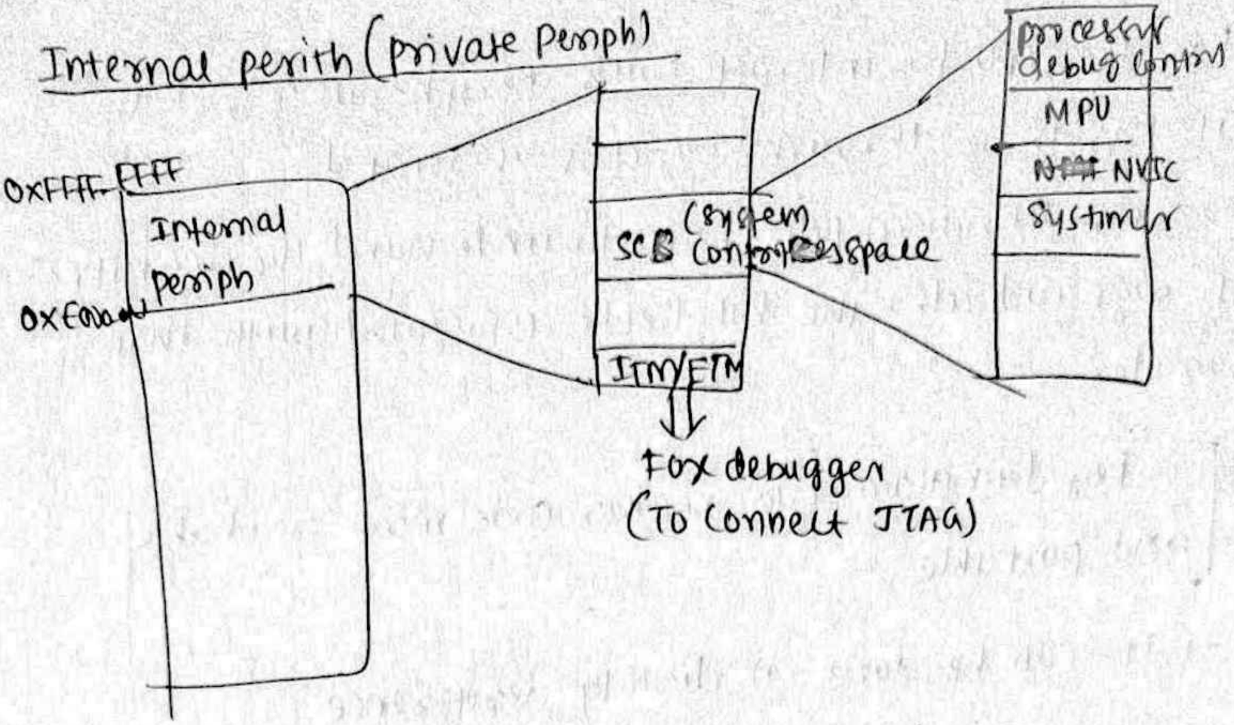
BASEPRI

Defines min priority for exception processing

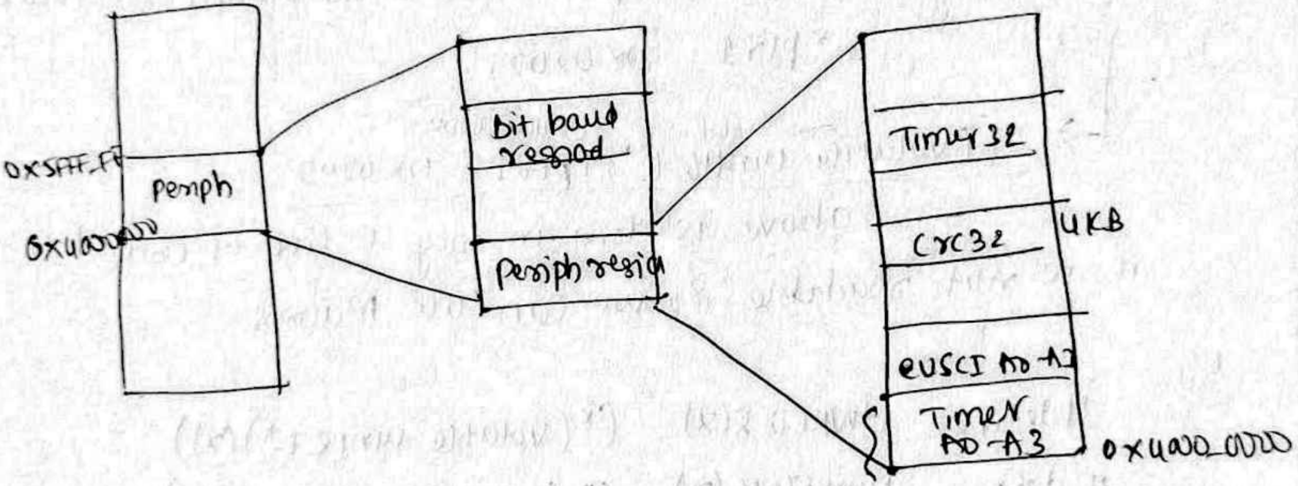
⊛ Control reg



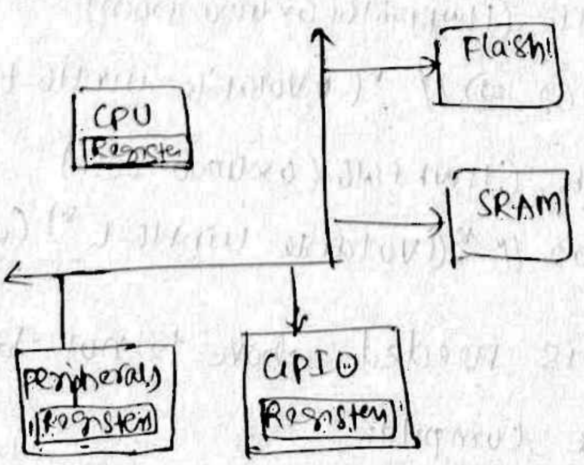
Internal perith (private Perph)



General periph registers



Register definition Files



→ For the cpu to interact with peripherals ~~registers~~. it has to be through registers associated.

→ Register definition Files helps to understand the definition of regs and also the bit fields associated with that register

> Reg definition File makes code more readable and portable

→ It can be done 1) directly dereference
2) Structure overlay

Eg:-

```
volatile uint16_t *ptr1 = (volatile uint16_t *)0x4000_0000
*ptr1 = 0x0202;

               0x4000_0000
*((volatile uint16_t *)) ~~ptr1~~ = 0x0202
```

above is done in only 1 line of code but it is not readable so we can use Macros

Eg:-

```
#define HWREG8(x)   (*(volatile uint8_t*)(x))
#define HWREG16(x)  (*(volatile uint16_t*)(x))
#define HWREG32(x)  (*(volatile uint32_t*)(x))

#define TA0CTL (HWREG16(0x4000_0000))
TA0CTL=0x0202 ⇒ // *((volatile uint16_t *)(0x4000_0000))=
                                                    0x0202
#define TA0R (HWREG16 (0x4000_0010)
TA0R= 0x0202 // *((volatile uint16_t *) (0x4000_0010))=0x0202
```

Note:- "volatile" is needed above to not to optimise the code by the compiler.