Embedded Computer Systems and Applications

Class #2: Data Processing

Chapters 8, 9, 12, 14, and 24

Instructor: Ken Q. Yang

Dept. of ECE, URI

Big Data and Applications

Advances in various branches of technology

- data sensing
- data communication
- data computation, and
- data storage

are driving an era of unprecedented innovation for information retrieval.

- Huge amount of data collected is empowering analytics and decision making process
- Examples:
  - Wal-Mart processes one million transactions an hour, and from these transactions they collect 2.5 petabytes of information
  - Facebook users upload 100 terabytes of data and 1 Billion pieces of content every day
  - Big data analytics hold the promise of supporting a wide range of healthcare functions.

Technology Constraints

- **The semiconductor industry has reached its physical scaling limits;**
- **Physical constraints, such as power and density, have therefore become the limiting factor**
- **Big data applications have a high DRAM memory bandwidth demand compared with standard CPU desktop and parallel applications.**
- **the rate of computation to memory access is relatively low compared with standard desktop**
- **Data movements among components of systems present challenge in computing**

Embedded computers show great advantages in terms of processing power, energy consumption, moving data around, and communication speed

Characterization of Big Data

- Big data applications are characterized by four critical features, referred as the four "Vs",

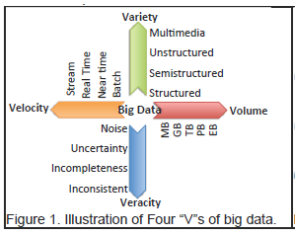


Figure 1. Illustration of Four "V"s of big data.

Typical Cloud Environment

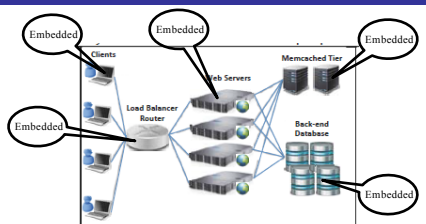
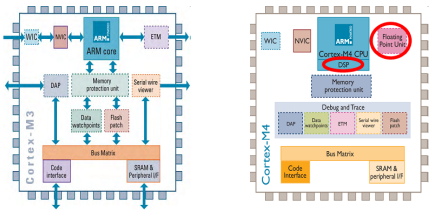


Figure 2. Design of a web-serving architecture

Computation Power

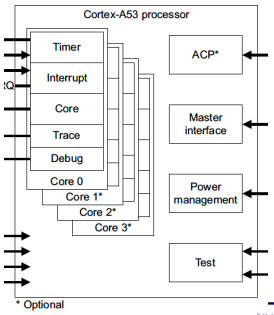
Cortex™-M3 → Cortex™-M4



Key Cortex™-M4 enhancements over Cortex-M3:

DSP instructions, SIMD instructions, optional floating point unit

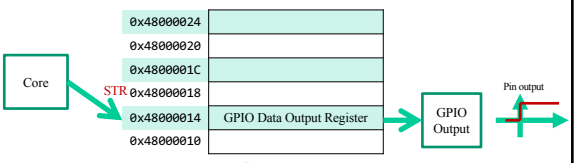
Cortex A53 Processor Architecture



7

Interfacing Peripherals

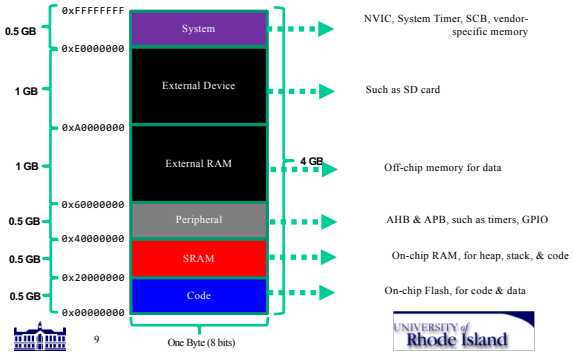
- **Port-mapped I/O**
  - Use special CPU instructions: `Special_instruction Reg, Port`
- **Memory-mapped I/O**
  - A simpler and more convenient way to interface I/O devices
  - Each device registers is assigned to a memory address in the address space of the microprocessor
  - Use native CPU load/store instructions: `LDR/STR Reg, [Reg, #imm]`



8

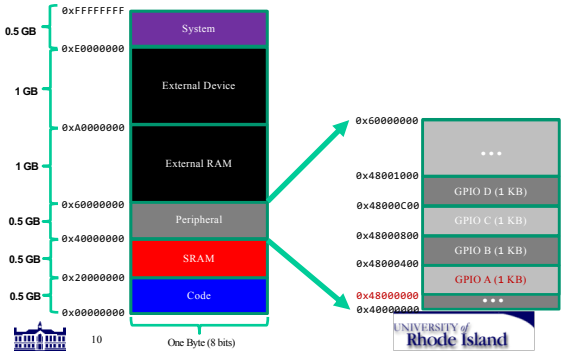
ARM Cortex-M microprocessors use memory-mapped I/O.

Memory Map of Cortex-M4



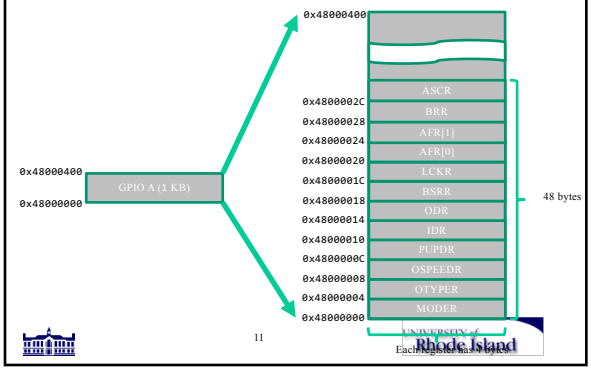
9

Memory Map of STM32L4



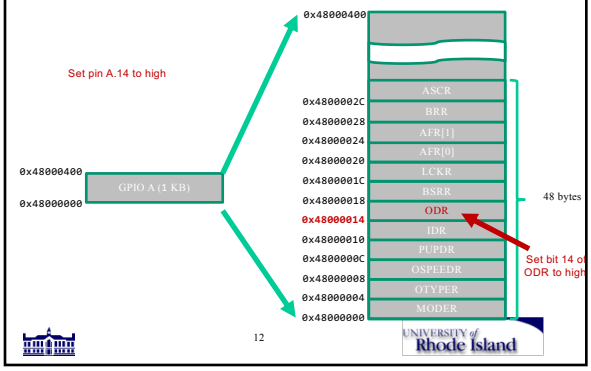
10

GPIO Memory Map



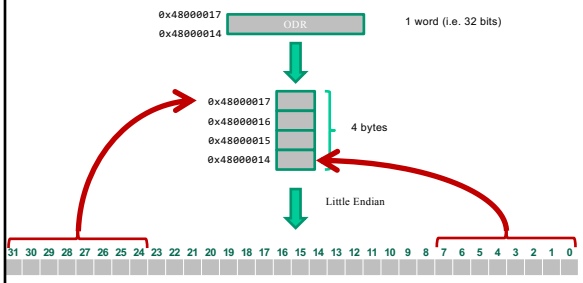
11

GPIO Memory Map

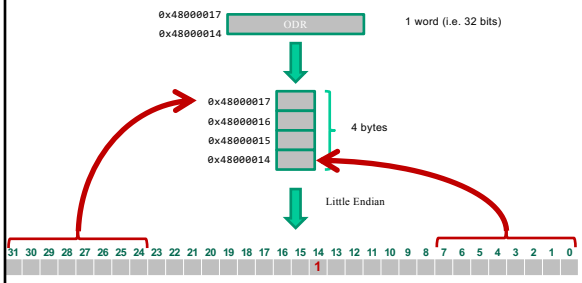


12

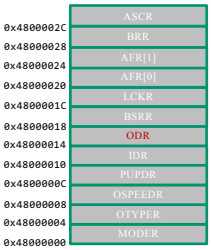
Output Data Register (ODR)



Output Data Register (ODR)



Dereferencing a Memory Address




```
typedef struct {
  volatile uint32_t MODER;    // Mode register
  volatile uint32_t OTYPER;   // Output type register
  volatile uint32_t OSPEEDR;  // Output speed register
  volatile uint32_t PUPDR;    // Pull-up/pull-down register
  volatile uint32_t IDR;      // Input data register
  volatile uint32_t ODR;      // Output data register
  volatile uint32_t BSRR;     // Bit set/reset register
  volatile uint32_t LCKR;     // Configuration lock register
  volatile uint32_t AFR[2];   // Alternate function registers
  volatile uint32_t BRR;      // Bit Reset register
  volatile uint32_t ASCR;     // Analog switch control register
} GPIO_TypeDef;

// Casting memory address to a pointer
#define GPIOA ((GPIO_TypeDef *) 0x48000000)
```

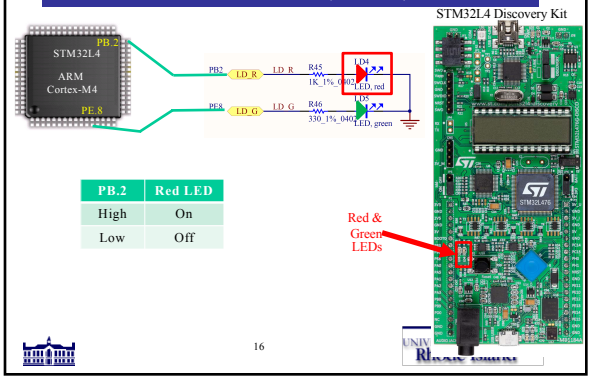
```
GPIOA->ODR |= 1UL<<14;
```

or

```
(*GPIOA).ODR |= 1UL<<14;
```

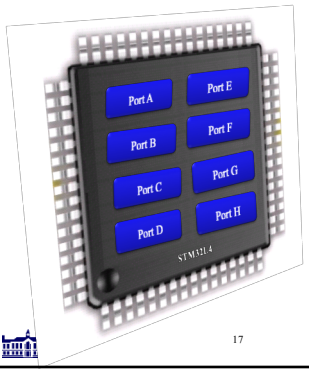

Red LED (PB.2)

STM32L4 Discovery Kit



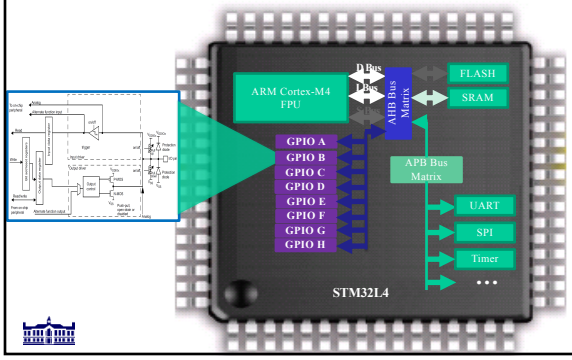
PB.2	Red LED
High	On
Low	Off

General Purpose Input/Output



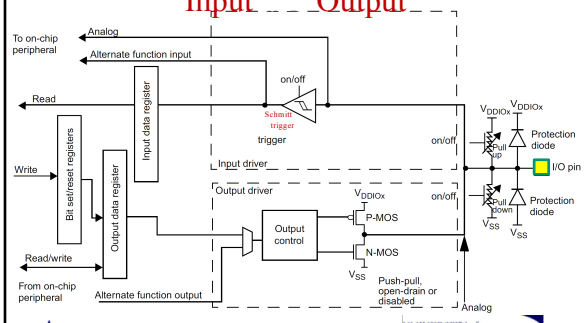
- 8 GPIO Ports:
  A, B, C, D, E, F, G, H
- Up to 16 pins in each port

General Purpose Input/Output

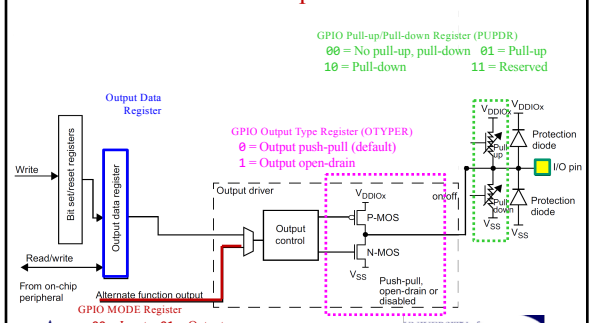


Basic Structure of an I/O Port Bit

Input Output



Basic Structure of an I/O Port Bit: Output



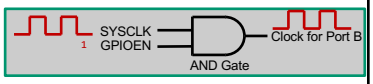
Enable Clock

- AHB2 peripheral clock enable register (RCC_AHB2ENR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RNG EN	Res.	AESEN
													rw		rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	ADCEN	OTGFS EN	Res.	Res.	Res.	Res.	GPIOH EN	GPIOG EN	GPIOF EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
		rw	rw					rw	rw	rw	rw	rw	rw	rw	rw

Bit 1 **GPIOBEN:** IO port B clock enable
Set and cleared by software.
0: IO port B clock disabled
1: IO port B clock enabled




```
#define  RCC_AHB2ENR_GPIOBEN  ((uint32_t)0x00000002U)

RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
```


GPIO Mode Register (MODER)

- 32 bits (16 pins, 2 bits per pin)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Pin 2 Pin 1 Pin 0

Bits 2y+1:2y **MODEy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

```
GPIOB->MODER &= ~(3UL<<4);  // Clear bits 4 and 5 for Pin 2
GPIOB->MODER |=   1UL<<4;   // Set bit 4, set Pin 2 as output
```

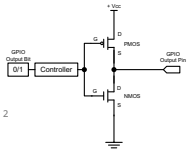

GPIO Output Type Register (OTYPE)

- 16 bits reserved, 16 data bits, 1 bit for each pin

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

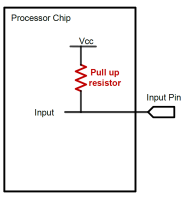
Bits 15:0 **OTy:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output type.
0: Output push-pull (reset state)
1: Output open-drain




```
GPIOB->OTYPE &= ~(1UL<<2);  // Clear bit 2
```

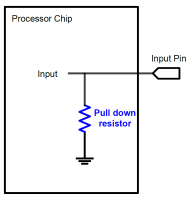

GPIO Input:

- A digital input can have three states: High, Low, and High-Impedance (also called floating, tri-stated, HiZ)



Pull-Up

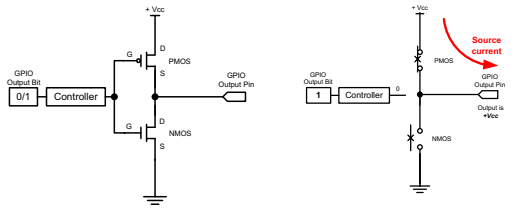
If external input is HiZ, the input is read as a valid HIGH.



Pull-Down

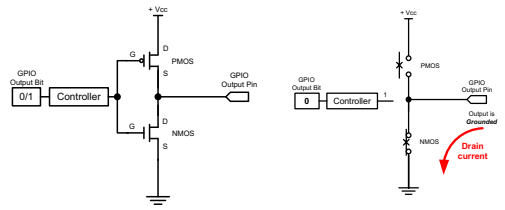
If external input is HiZ, the input is read as a valid LOW.

GPIO Output:



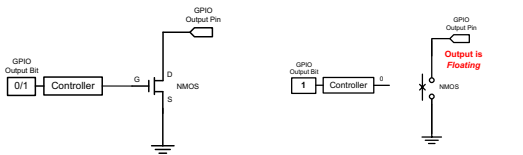
GPIO Output = 1
Source current to external circuit

GPIO Output:



GPIO Output = 0
Drain current from external circuit

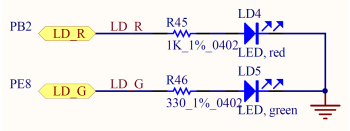
GPIO Output:



Output = 1
GPIO Pin has high-impedance to external circuit

GPIO Output: Push-Pull *vs* Open-Drain

Output Bit	Push-Pull	Open-Drain
1	High	HiZ
0	Low	Low



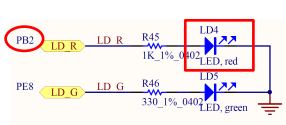
Use push-pull output, instead of open-drain output!

GPIO Output Data Register (ODR)

- 16 bits reserved, 16 data bits, 1 bit for each pin

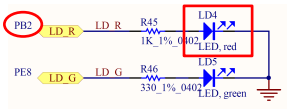
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Pin 2




```
GPIOB->ODR |= 1UL << 2;    // Set bit 2
```


Light up the Red LED (PB.2)




```
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;  // Enable clock of Port B

GPIOB->MODER &= ~(3UL<<4); // Clear mode bits
GPIOB->MODER |=   1UL<<4;  // Set mode to output

GPIOB->OTYPE &= ~(1UL<<2); // Select push-pull output

GPIOB->ODR   |= 1UL << 2;  // Output 1 to turn on red LED
```

GPIO Initialization

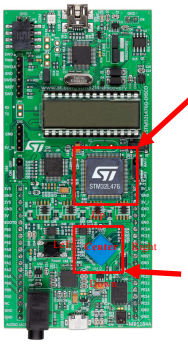
- Turn on the clock to the GPIO Port (e.g. Port B)

```
RCC->AHBENR |= RCC_AHBENR_GPIOBEN; Reset and Clock Control (RCC)
```

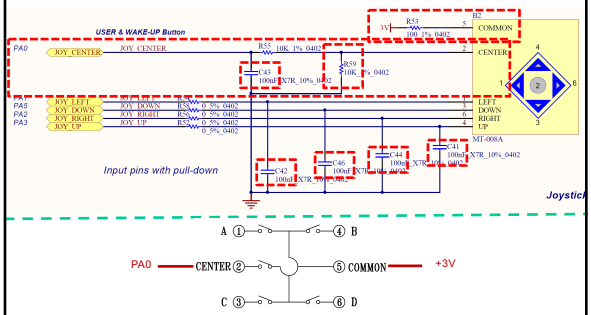
- Configure GPIO mode, output type, speed, pull-up/pull-down

```
typedef struct
{
  __IO uint32_t MODER;
  __IO uint16_t OTYPER;
  uint16_t RESERVED0;
  __IO uint32_t OSPEEDR;
  __IO uint32_t PUPDR;
  __IO uint16_t IDR;
  uint16_t RESERVED1;
  __IO uint16_t ODR;
  uint16_t RESERVED2;
  __IO uint16_t BSRRL; /* BSRR register is split to 2 * 16-bit fields BSRRL */
  __IO uint16_t BSRRH; /* BSRR register is split to 2 * 16-bit fields BSRRH */
  __IO uint32_t LCKR;
  __IO uint32_t AFR[2];
} GPIO_TypeDef;
#define PERIPH_BASE           ((uint32_t)0x40000000)
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)
#define GPIOB_BASE            (AHBPERIPH_BASE + 0x0400)
#define GPIOB                 ((GPIO_TypeDef *) GPIOB_BASE)
```

Joystick

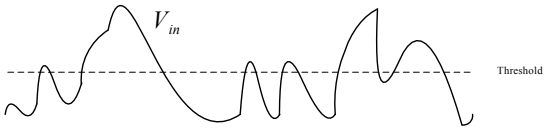


Joystick



Internal diagram of joystick

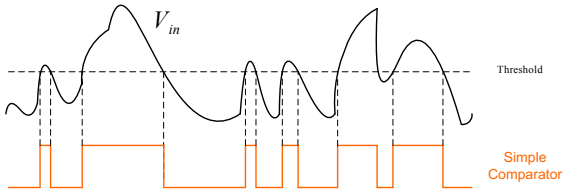
Schmitt Trigger



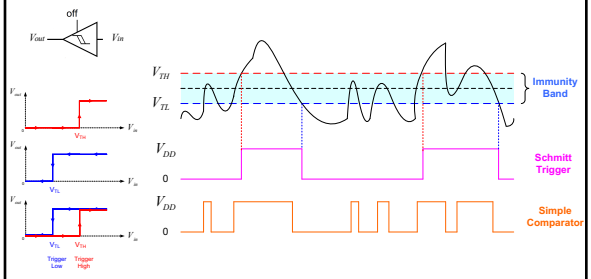
Analog signals

- Noisy
- Rise and fall slowly (small slew rate)

Schmitt Trigger

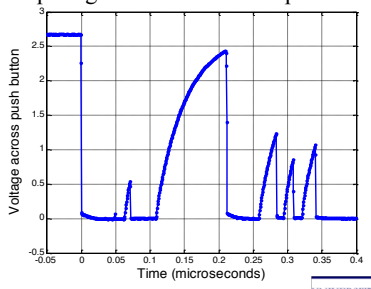


Schmitt Trigger

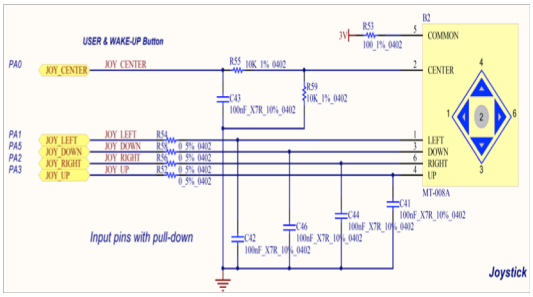


I/O Debouncing

- Example signal when a button is pressed



HW RC Debouncer



Structured Programming, Cont'd

Calling a Subroutine

BL *label*

- Step 1: LR = PC + 4
- Step 2: PC = label

- Notes:
  - *label* is name of subroutine
  - Compiler translates label to memory address
  - After call, LR holds return address (the instruction following the call)

Caller Program

```
MOV r4, #100
...
BL  foo
...
```

Subroutine/Callee

```
foo PROC
    ...
    MOV   r4, #10
    ...
    BX    LR
    ENDP
```


Exiting a Subroutine

BX *LR*

- PC = LR

Caller Program

```
MOV r4, #100
...
BL  foo
...
```

Subroutine/Callee

```
foo PROC
    ...
    MOV   r4, #10
    ...
    BX    LR
    ENDP
```


Recursive Functions

- A recursive function is one that solves its task by calling itself on smaller pieces of data.
- An effective tactic is to
  - divide a problem into sub-problems of the same type as the original,
  - solve those sub-problems, and
  - combine the results

Defining Factorial(n)

Product of the first n numbers

$$1 \times 2 \times 3 \times \ldots \times n$$

```
factorial(0)        = 1
factorial(1)        = 1            = 1 × factorial(0)
factorial(2)      = 2×1            = 2 × factorial(1)
factorial(3)    = 3×2×1            = 3 × factorial(2)
factorial(4)  = 4×3×2×1            = 4 × factorial(3)
factorial(n)    = n×(n-1)×…×1        = n × factorial(n-1)
```

Classic Example: Factorial

- Factorial is the classic example:
  - 6! = 6 × 5!
  - 6! = 6 × 5 x 4!
  - …
  - 6! = 6 × 5 × 4 × 3 × 2 × 1
- The factorial function can be easily written as a recursive function:

```
int Factorial(int n) {

    if (n < 2)
        return 1; /* base case */

    return (n * Factorial(n - 1));

}
```


Recursion vs Iteration

Any problem that can be solved recursively can also be solved iteratively (using loop).

Recursive functions vs Iterative functions

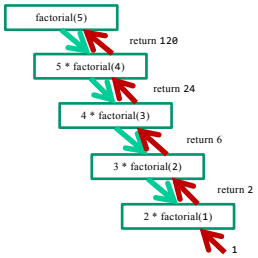
- Cons:
  - Recursive functions are slow
  - Recursive function take more memory
- Pros
  - Recursive functions resembles the problem more naturally
  - Recursive function are easier to program, and debug.

Recursive Factorial in C

```
int factorial(int n);

int main(void){
  factorial(5);
  return 0;
}

int factorial(int n) {
  int f;
  if(n==1)
    return 1;
  else
    f = n * factorial(n-1);
  return f;
}
```



Recursive Functions

- push LR (& working registers) onto stack before nested call
- pop LR (& working registers) off stack after nested return

Recursive Factorial in Assembly

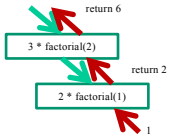
```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
                MOV     r0, #0x03
0x08000130      BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```



Recursive Factorial in Assembly

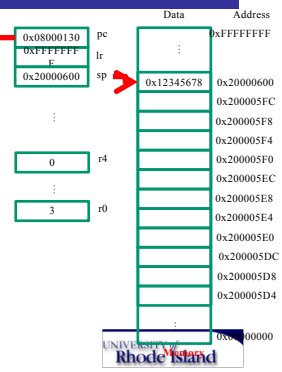
```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
                MOV     r0, #0x03
0x08000130      BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```



Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
                BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000136	pc	⋮	0xFFFFFFFF
0x08000134	lr	0x12345678	0x20000600
0x20000600	sp		0x200005FC
			0x200005F8
			0x200005F4
0	r4		0x200005F0
			0x200005EC
3	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

49

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
                BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000138	pc	⋮	0xFFFFFFFF
0x08000134	lr	0x12345678	0x20000600
0x200005F8	sp	0x08000134	0x200005FC
		0	0x200005F8
			0x200005F4
3	r4		0x200005F0
			0x200005EC
3	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

Order in which registers are specified is not important: the lowest register is always stored at the lowest address

50

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
                BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000144	pc	⋮	0xFFFFFFFF
0x08000134	lr	0x12345678	0x20000600
0x200005F8	sp	0x08000134	0x200005FC
		0	0x200005F8
			0x200005F4
3	r4		0x200005F0
			0x200005EC
2	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

51

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
                BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000136	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F8	sp	0x08000134	0x200005FC
		0	0x200005F8
			0x200005F4
3	r4		0x200005F0
			0x200005EC
2	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

52

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
                BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000138	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F0	sp	0x08000134	0x200005FC
		0	0x200005F8
		0x08000148	0x200005F4
2	r4	3	0x200005F0
			0x200005EC
2	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

53

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
                BL      factorial
0x08000134 stop B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140 loop POP     {r4, pc}
0x08000142 NZ   SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000144	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F0	sp	0x08000134	0x200005FC
		0	0x200005F8
		0x08000148	0x200005F4
2	r4	3	0x200005F0
			0x200005EC
1	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

54

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
loop
0x08000142      SUBS    r0, r4, #1
NZ              BL      factorial
0x08000144      MUL     r0, r4, r0
0x08000148      B       loop
0x0800014C

                END
```

		Data	Address
0x08000136	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F0	sp	0x08000134	0x200005FC
⋮		0	0x200005F8
2	r4	0x08000148	0x200005F4
⋮		3	0x200005F0
1	r0		0x200005EC
			0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

55

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
loop
0x08000142      SUBS    r0, r4, #1
NZ              BL      factorial
0x08000144      MUL     r0, r4, r0
0x08000148      B       loop
0x0800014C

                END
```

		Data	Address
0x08000138	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005E8	sp	0x08000134	0x200005FC
⋮		0	0x200005F8
1	r4	0x08000148	0x200005F4
⋮		3	0x200005F0
1	r0	0x08000148	0x200005EC
		2	0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

56

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
loop
0x08000142      SUBS    r0, r4, #1
NZ              BL      factorial
0x08000144      MUL     r0, r4, r0
0x08000148      B       loop
0x0800014C

                END
```

		Data	Address
0x0800013E	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005E8	sp	0x08000134	0x200005FC
⋮		0	0x200005F8
1	r4	0x08000148	0x200005F4
⋮		3	0x200005F0
1	r0	0x08000148	0x200005EC
		2	0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

57

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
loop
0x08000142      SUBS    r0, r4, #1
NZ              BL      factorial
0x08000144      MUL     r0, r4, r0
0x08000148      B       loop
0x0800014C

                END
```

		Data	Address
0x08000148	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F0	sp	0x08000134	0x200005FC
⋮		0	0x200005F8
2	r4	0x08000148	0x200005F4
⋮		3	0x200005F0
1	r0	0x08000148	0x200005EC
		2	0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

58

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
loop
0x08000142      SUBS    r0, r4, #1
NZ              BL      factorial
0x08000144      MUL     r0, r4, r0
0x08000148      B       loop
0x0800014C

                END
```

		Data	Address
0x08000148	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F0	sp	0x08000134	0x200005FC
⋮		0	0x200005F8
2	r4	0x08000148	0x200005F4
⋮		3	0x200005F0
2	r0		0x200005EC
			0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

59

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
loop
0x08000142      SUBS    r0, r4, #1
NZ              BL      factorial
0x08000144      MUL     r0, r4, r0
0x08000148      B       loop
0x0800014C

                END
```

		Data	Address
0x08000140	pc	⋮	0xFFFFFFFF
0x08000148	lr	0x12345678	0x20000600
0x200005F0	sp	0x08000134	0x200005FC
⋮		0	0x200005F8
2	r4	0x08000148	0x200005F4
⋮		3	0x200005F0
2	r0		0x200005EC
			0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
		⋮	0x00000000

Order in which registers are specified is not important.

60

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
0x08000142      SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000148	pc		0xFFFFFFFF
0x08000148	lr		
0x200005F8	sp	0x12345678	0x20000600
		0x08000134	0x200005FC
		0	0x200005F8
			0x200005F4
3	r4		0x200005F0
			0x200005EC
2	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
			0x00000000

61

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
0x08000142      SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000148	pc		0xFFFFFFFF
0x08000148	lr		
0x200005F8	sp	0x12345678	0x20000600
		0x08000134	0x200005FC
		0	0x200005F8
			0x200005F4
3	r4		0x200005F0
			0x200005EC
6	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
			0x00000000

62

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
0x08000142      SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000148	pc		0xFFFFFFFF
0x08000148	lr		
0x200005F8	sp	0x12345678	0x20000600
		0x08000134	0x200005FC
		0	0x200005F8
			0x200005F4
3	r4		0x200005F0
			0x200005EC
6	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
			0x00000000

63

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
0x08000142      SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

		Data	Address
0x08000134	pc		0xFFFFFFFF
0x08000148	lr		
0x20000600	sp	0x12345678	0x20000600
			0x200005FC
			0x200005F8
			0x200005F4
0	r4		0x200005F0
			0x200005EC
6	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
			0x00000000

64

Recursive Factorial in Assembly

```
                AREA    main, CODE, READONLY
                EXPORT  __main

                ENTRY

__main          PROC
0x08000130      MOV     r0, #0x03
0x08000134      BL      factorial
stop            B       stop
                ENDP

factorial
0x08000136      PUSH    {r4, lr}
0x08000138      MOV     r4, r0
0x0800013A      CMP     r4, #0x01
0x0800013C      BNE     NZ
0x0800013E      MOVS    r0, #0x01
0x08000140      POP     {r4, pc}
0x08000142      SUBS    r0, r4, #1
0x08000144      BL      factorial
0x08000148      MUL     r0, r4, r0
0x0800014C      B       loop

                END
```

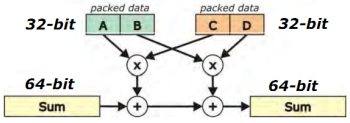
		Data	Address
0x08000134	pc		0xFFFFFFFF
0x08000148	lr		
0x20000600	sp	0x12345678	0x20000600
			0x200005FC
			0x200005F8
			0x200005F4
0	r4		0x200005F0
			0x200005EC
6	r0		0x200005E8
			0x200005E4
			0x200005E0
			0x200005DC
			0x200005D8
			0x200005D4
			0x00000000

65

SIMD(Single instruction Multiple data) Operations

SIMD extensions perform multiple operations in one cycle

Sum = Sum + (A x C) + (B x D)



SIMD techniques operate with packed data

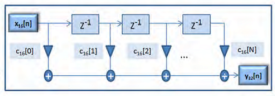
66

DSP Example: Cortex™-M4 FIR

$$y_{32}[n] = \sum_{i=0}^{N} x_{16}[n-i] * c_{16}[i]$$

y_{32} – filter ouput (32 bit); N - filter order

x_{16} – 16 bit input data; c_{16} – 16 bit filter coeff



Cortex-M3 Code Segment:

```
FIR_LOOP:
LDR     R2,[R0],#4    ;(2) Load input x16
LDR     R3,[R1],#4    ;(2) Load coeff c16
SXTH    R4, R2        ;(1) Extract x16[n-i]
ASR     R2, R2,#16    ;(1) Extract x16[n-i-1]
SXTH    R5, R3        ;(1) Extract c16[i]
ASR     R3, R3,#16    ;(1) Extract c16[i+1]
MLA     R6, R4, R5    ;(2) y32 += x16[n-i]*c16[i]
MLA     R6, R2, R3    ;(2) y32 += x16[n-i-1]*c16[i+1]
SUBS    R7, R7, #2    ;(1) loop count -= 2
BNE     FIR_LOOP      ;(2)

Note:
1. In these examples, FIR_LOOP is unrolled by 2
2. This example assumes number of taps is even.
```

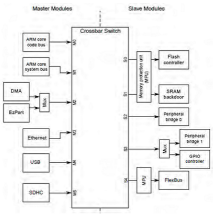
Cortex-M4 Code Segment:

```
FIR_LOOP:
LDR     R2,[R0],#4 ;(1) Load input x16,x16[n-1]
LDR     R3,[R1],#4 ;(2) Load coeff c16[i], c16[i+1]
SUBS    R5, R5, #2 ;(1) loop count -= 2
SMLAD   R4, R2, R3 ;(1) y32 +=x16[n-1,n-1-1]*C16[i,i+1]
BNE     FIR_LOOP   ;(2)
```

Processor	Kernel cycles	Total Cycles	Number of Instructions	Register usage
Cortex-M3	8	15	10	7
Cortex-M4	1	7	5	5
Advantage	**8x**	**~2.2x**	**2x**	**1.4x**

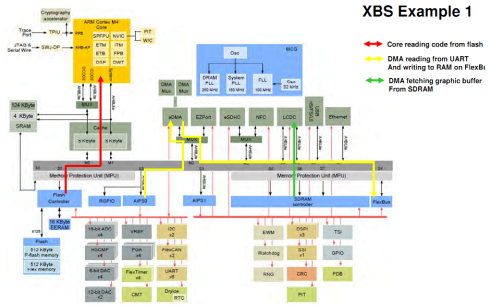
Example (non-binding) from Ittiam Systems, a leading provider of signal processing software on ARM platforms

Cross Bar Switch Configuration



- Symmetric crossbar bus switch implementation
  - Allows concurrent accesses from different masters to different slaves
  - Slave arbitration attributes configured on a slave by slave basis
- 32 -bit wide and supports byte, 2 byte, 4 byte, and 16 byte burst transfers
- Operates at a 1-to-1 clock frequency with the bus masters
- Low-power park mode support

XBS Example



Memory Bandwidth