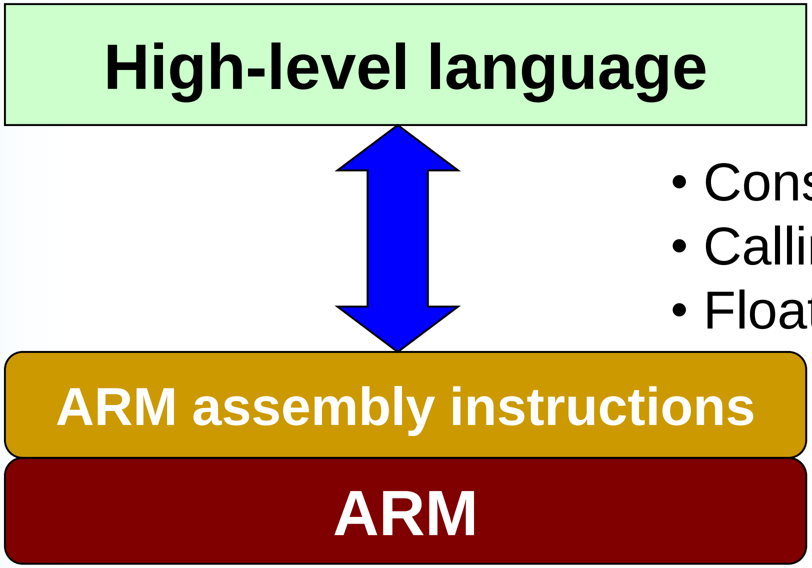
Architectural Support for High-Level Language

Peng-Sheng Chen

Fall, 2017

Introduction

- Look at the requirements that a high-level language imposes on an architecture
- See how those requirements may be met



- Constructs of high-level languages
- Calling convention
- Floating-point support

Outline

- Abstraction in software design
- Data types
- Floating-point data types
- Expressions
- Conditional statements
- Loops
- Functions and procedures
- Use of Memory
- Run-time environment

Outline

- **Abstraction in software design**
- Data types
- Floating-point data types
- Expressions
- Conditional statements
- Loops
- Functions and procedures
- Use of Memory
- Run-time environment

Abstraction in Software Design

- **Determine the higher levels of abstraction**
  - Simplify the program design
  - High-level language
- **Assembly-level abstraction**
  - Work directly with the raw machine instructions
  - Express the program by instructions, addresses, registers, …

Outline

- Abstraction in software design
- **Data types**
- Floating-point data types
- Expressions
- Conditional statements
- Loops
- Functions and procedures
- Use of Memory
- Run-time environment

Data Types (1)

- Numbers
- Roman numerals
- Decimal numbers
- BCD (binary coded decimal)
- Binary notation
- Hexadecimal notation
- Number ranges
- Signed integers
- Other number sizes
- Real numbers
- Printable characters

Data Types (2)

- ASCII
- **ARM support for character**
  - Unsigned byte load / store instructions
- Byte ordering
  - Character encode

1	9	9	5

  - Read / Store a 32-bit word
  - little- or big-endian

Data Types (3)

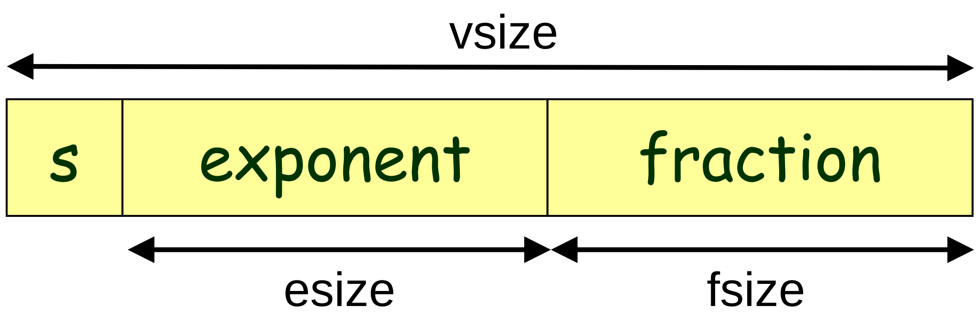
- High-level languages
- ANSI C basic data types
  - character, short integers, integer, long, …
  - **ARM C compiler**
    - **unsigned integer: 32 bits**
    - **unsigned long integer: 32 bits**
    - **unsigned short integer: 16 bits**
- ANSI C derived data types
  - Array, functions, structures, …
- ARM architectural support for C data types

Outline

- Abstraction in software design
- Data types
- **Floating-point data types**
- The ARM floating-point architecture
- Expressions
- Conditional statements
- Loops
- Functions and procedures
- Use of Memory
- Run-time environment

Floating-Point Data Types

- **IEEE-754**



  - v = (-1)^s x 2^exponent x (1.fraction)
  - Single: esize = 8, fsize = 23, vsize = 32
  - Double: esize = 11, fsize = 52, vsize = 64
  - Double extended, vsize > 64

IEEE 754 Single Precision Representation of ‘1995’

31	30 … 23	22 … 0
0	1 0 0 0 1 0 0 1	1 1 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0

```
1995 = 11111001011
     = 1.1111001011 x 2^1010

The exponent is 127 + 10 = 137
```

value = $(-1)^S \times 1.\text{fraction} \times 2^{(\text{exponent}-127)}$

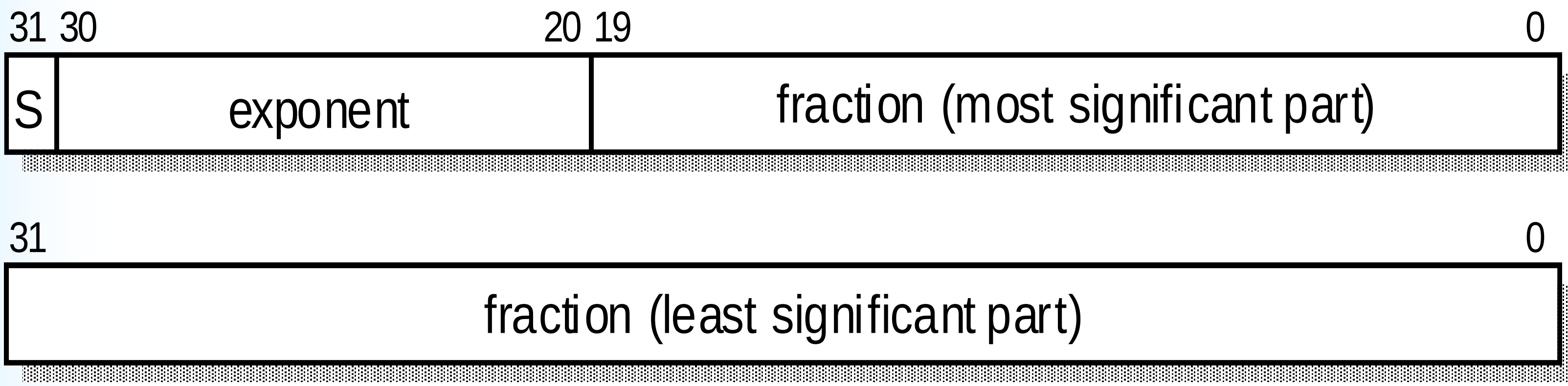
Reserved Numbers in IEEE 754 (1)

- The exponent is either zero or 255
- **Zero**
  - A zero exponent and fraction (positive zero and negative zero)
- **Plus / minus infinity**
  - The maximum exponent value
  - Zero fraction

Reserved Numbers in IEEE 754 (2)

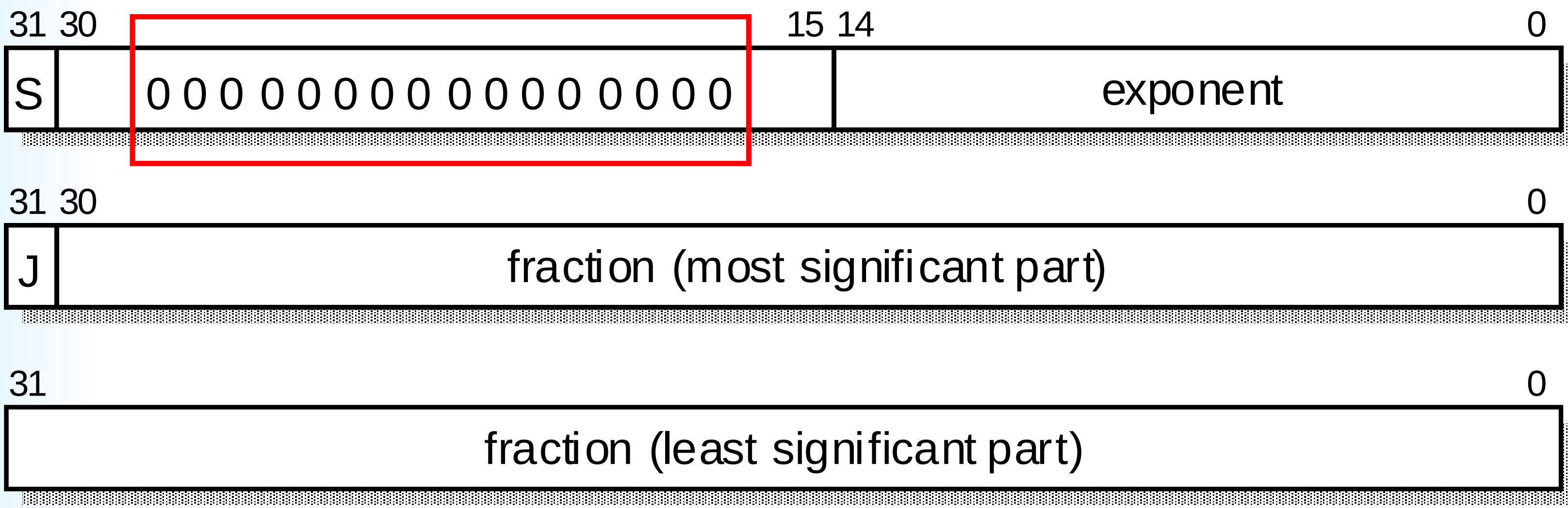
- **NaN (Not a Number)**
  - The maximum exponent value
  - Non-zero fraction
- **Denormalized number**
  - The number are too small to normalize within this format
  - Zero exponent
  - Non-zero fraction

IEEE 754 Double Precision Floating-Point Number Format



IEEE 754 Double Extended Precision Floating-Point Number Format

80 bits of information spread across three words



IEEE 754 Packed Decimal Floating-Point Number Format

31	30	29	28	27 … 12	11 … 0
E	D	0	0	exponent (4 BCD digits)	dec. (3 most sig. BCD digs.)

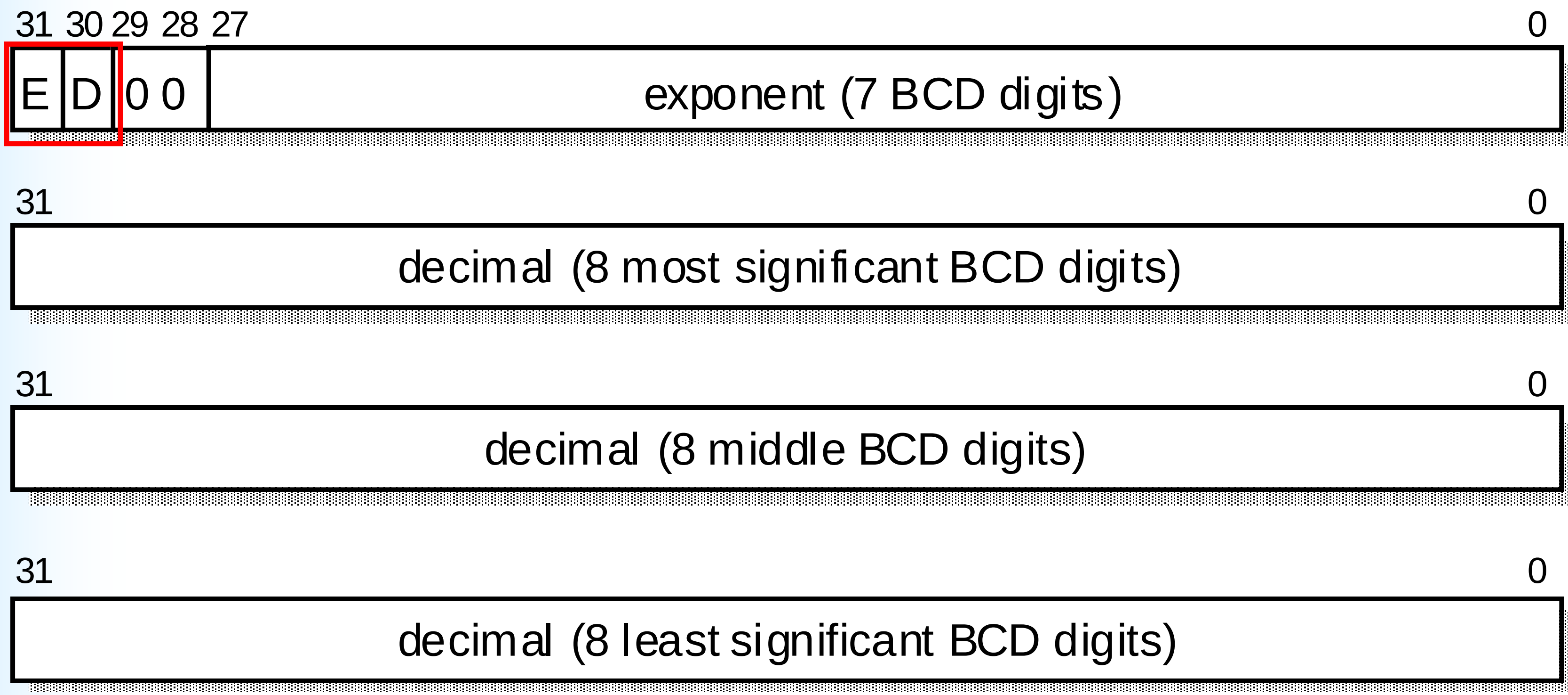
31 … 0
decimal (8 middle BCD digits)

31 … 0
decimal (8 least significant BCD digits)

value = $$(-1)^{D} \times \text{decimal} \times 10^{((-1)^{E} \times \text{exponent})}$$

3 words = 96 bits

IEEE 754 Extended Packed Decimal Floating-Point Number Format



4 words = 128 bits

Outline

- Abstraction in software design
- Data types
- Floating-point data types
- **Expressions**
- Conditional statements
- Loops
- Functions and procedures
- Use of Memory
- Run-time environment

Expressions

- Register use
  - Compilers help to allocate
- ARM support
  - 3 address format is good for compilers
- Pointer arithmetic
- Arrays
  - Ex:

```
int *p;
int i = 1;
p = p + i;
```

Assume: p in r0, i in r1

```
ADD   r0, r0, r1, LSL #2  ; scale r1 to int
```

Accessing Operands

- Pass an argument via a register or stack
- A constant => in the procedure's literal pool
- A local variable
  - Allocated space on the **stack**
- As a global variable
  - Allocated space in the **static area**

Outline

- Abstraction in software design
- Data types
- Floating-point data types
- Expressions
- **Conditional statements**
- Loops
- Functions and procedures
- Use of Memory
- Run-time environment

Conditional Statements (1)

- if … else

```
if (a > b)
    c = a;
else
    c = b;
```

```
CMP     r0, r1   ; if (a>b)
MOVGT   r2, r0   ; c = a
MOVLE   r2, r1   ; c = b
```

```
MOV     r2, r0   ; c = a
CMP     r0, r1   ; if (a>b)
MOVLE   r2, r1   ; c = b
```

For the case with simple “if” statements

Conditional Statements (2)

- A complex "if .. else" example

```
if (a > b) {
    c = a;
    stmt 1;
    …
} else {
    c = b;
    stmt 2;
    …
}
```

```
        CMP    r0, r1    ; if (a>b)
        BLE    ELSE
        MOV    r2, r0    ; c = a
        …                ; stmt 1
        …                ; …
        B      ENDIF
ELSE    MOV  r2, r1      ; c = a
        …                ; stmt 2
        …                ; …
ENDIF
```

Conditional Statements: switch…case (1)

- 假設所要執行之不同的動作依賴於某個變數x
- 0 <= x < N

```
int ref_switch(int x)
{
    switch (x) {
    case 0: return method_0();
    case 1: return method_1();
    case 2: return method_2();
    case 3: return method_3();
    case 4: return method_4();
    case 5: return method_5();
    case 6: return method_6();
    case 7: return method_7();
    default: return method_d();
    }
}
```

Conditional Statements: switch…case (2)

- A programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program

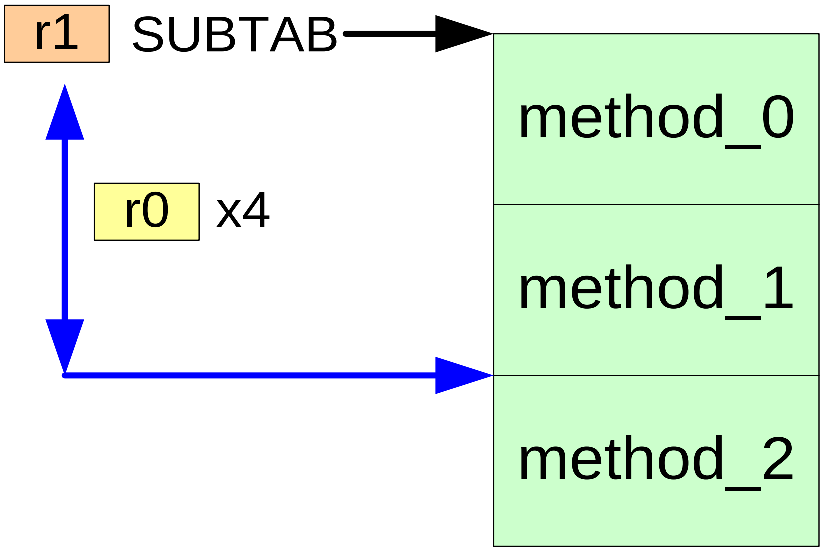
Note: slow when the list is long, and all subroutines are equally frequent

```
        BL      JUMPTAB
        ..
JUMPTAB
        CMP     r0, #0
        BEQ     method_0
        CMP     r0, #1
        BEQ     method_1
        CMP     r0, #2
        BEQ     method_2
        ..
```

Conditional Statements: switch…case (3)

- "**DCD**" ("**.word**") directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression in the right

```
    BL      JUMPTAB
    ..
JUMPTAB
    ADR     r1, SUBTAB
    CMP     r0, #SUBMAX
    LDRLS   pc, [r1, r0, LSL #2]
    B       method_d
SUBTAB
    DCD     method_0
    DCD     method_1
    DCD     method_2
    ..
```



Outline

- Abstraction in software design
- Data types
- Floating-point data types
- Expressions
- Conditional statements
- **Loops**
- Functions and procedures
- Use of Memory
- Run-time environment

Loops

- Three forms of loop-control structure
  - for loops
  - while loops
  - do…while loops

For Loops

```
for (i=0; i<10; i++) {
   a[i] = 0;      /* a[i] is an integer */
}
```

".a" is the address of a[0]

```
      MOV   r1, #0        ; The value to be stored in a[i]
      ADR   r2, .a        ; r2 points to a[0]
      MOV   r0, #0        ; i = 0
LOOP  CMP   r0, #10       ; i < 10 ?
      BGE   EXIT          ; if i >= 10 finish
      STR   r1, [r2, r0, LSL #2] ; a[i] = 0
      ADD   r0, r0, #1 ; i ++
      B     LOOP
EXIT  …
```

While Loops (1)

假設while construct繼續執行的條件是不相等

```
LOOP:   ...             ; evaluate exp
        BEQ   EXIT
        ...             ; loop body
        B     LOOP
EXIT:   ...
```

```
        B     TEST
LOOP:   ...             ; loop body
        ...
TEST:   ...             ; evaluate exp
        BNE   LOOP
EXIT:   ...
```

branch instruction移到最後面，loop body較無branch的干擾

While Loops (2)

```
        B     TEST
LOOP:   ...             ; loop body
        ...
TEST:   ...             ; evaluate exp
        BNE   LOOP
EXIT:   ...
```

```
        ...             ; evaluate exp
        BEQ   EXIT      ; skip loop if necessary
LOOP:   ...             ; loop body
        ...
        ...             ; evaluate exp
        BNE   LOOP
EXIT:   ...
```

Do… While Loops

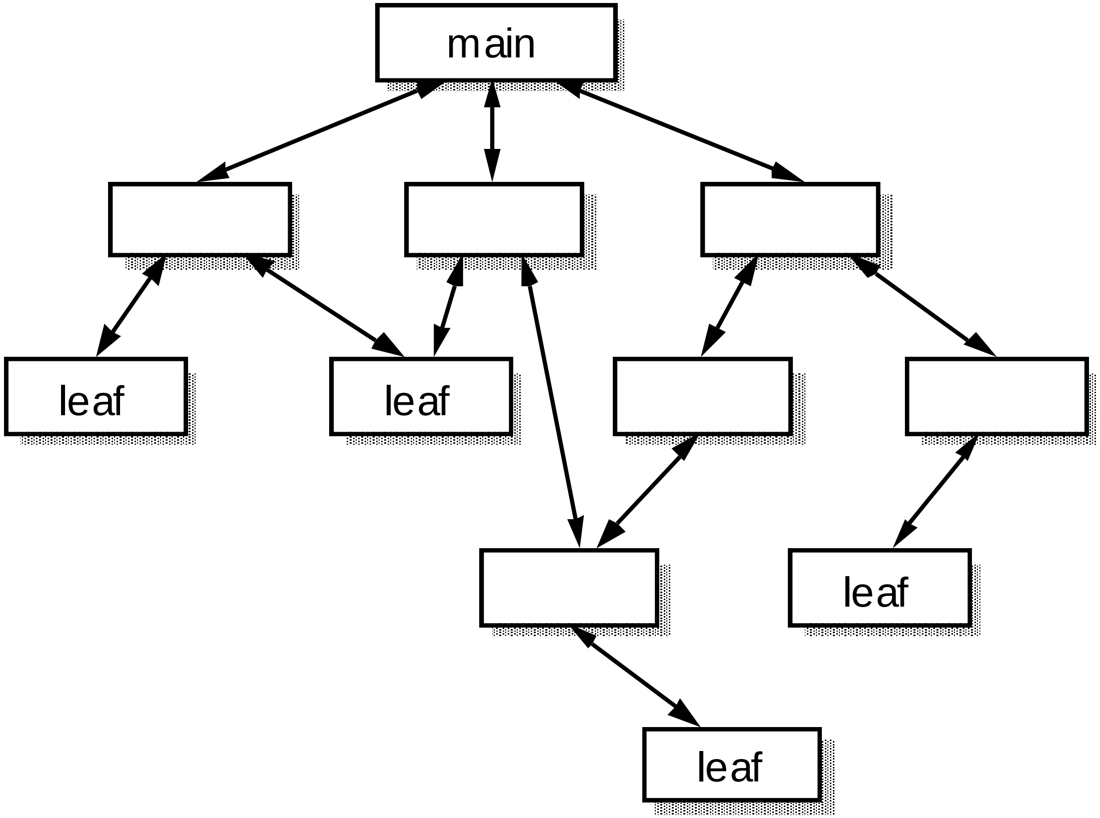
```
LOOP:   ...           ; loop body
        ...
        ...           ; evaluate exp
        BNE  LOOP
EXIT:   ...
```

Outline

- Abstraction in software design
- Data types
- Floating-point data types
- Expressions
- Conditional statements
- Loops
- **Functions and procedures**
- Use of Memory
- Run-time environment

Typical Hierarchical Program Structure

- Break down large programs into components that are small enough to be thoroughly tested



Terminology (1)

- **Subroutine**
  – A generic term for a routine that is called by a higher-level routine
- **Function**
  – A subroutine which returns a value through its name
  – Ex: c = max(a, b);
- **Procedure**
  – A subroutine which is called to carry out some operation on specified data items
  – Ex: printf("Hello World\n");

Terminology (2)

- **Arguments**
  - An expression passed to a function call
- **Parameters**
  - A value received by the function

```
void func(int a, int b)
{
    ...
}

int main(void)
{
    func(100,200);
    return 0;
}
```

parameters

arguments

Situation 1

```
main:

    ...
    bl func
    ...

    .end
```

```
func:

    ...
    ...
    ...

    .end
```

如果main function要傳遞一個integer到func function，要怎麼傳？

Situation 1

main:

func:

如果main function要傳遞一個integer到func function，要怎麼傳？

- 透過register r1
- 透過register r2
- …
- 透過stack
- 透過memory

Situation 1

main:

…
bl func

func:

…
…

- 如果main function與func function是同一個人寫的，則不會發生問題
- 如果main function與func function是不同的人寫的，則會有不知道對方是用什麼方式傳遞參數的問題
- 如果任何人都可以撰寫函式，則問題更複雜…

Situation 2

Caller	Callee

main:
 	
    use r1
    use r9
    bl func
    use r9
    use r1
    .end
	func:
	
	    write r1
	    write r9
	    …
	
	    .end

如果caller在呼叫func function之後還會用到呼叫func function之前的r1與r9的值，該怎麼辦？

Situation 2

Caller	Callee
main: use r1 use r9 bl func	func: write r1 write r9 …

如果caller在呼叫func function之後還會用到呼叫func function之前的r1與r9的值，該怎麼辦？

- Caller幫忙save
- Callee幫忙save

Situation 2

Caller

```
main:

    use r1
    use r9
    save r1 and r9
    bl func
    restore r1 and r9
    use r9
    use r1
    .end
```

Callee

```
func:

    write r1
    write r9
    …

    .end
```

Caller saved

Situation 2

Caller	Callee
main:	func:
use r1	save r1 and r9
use r9	write r1
bl func	write r9
use r9	…
use r1	restore r1 and r9
.end	.end

Callee saved

Situation 2

Caller	Callee
main: use r1 use r9 bl func	func: write r1 write r9 …

如果caller在呼叫func function之後還會用到呼叫func function之前的r1與r9的值，該怎麼辦？

- Caller幫忙save
- Callee幫忙save

如果caller與callee是不同的人寫的，那很難知道撰寫函式的人是否有先save register

ARM Procedure Call Standard (1)

- **Support flexible mixing of routines**
  - Generated by different compilers / different assemblers
  - Written in assembly language

Calling convention

- **ARM Limited defines a set of rules for procedure entry and exit**
  - **ARM Procedure Call Standard (APCS)**
  - 只要遵循**APCS**的規則，不同編譯器（人）所產生的**object code**，就可以相互呼叫，**link**在一起
  - **Assembly code**和**C program**可以交互參照

ARM Procedure Call Standard (2)

- Define particular use of general-purpose registers
- Define stack use from full/empty, ascending/descending choices
- Define the format of a stack-based data structure used for back-tracing when debugging programs
- Define the function argument and result passing mechanism to be used by all externally visible functions and procedures
- Support the ARM shared library mechanism

APCS Register Use Convention (1)

Register	Synonym	Special	Role in the procedure call standard
r15		**PC**	The Program Counter.
r14		**LR**	The Link Register.
r13		**SP**	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
r10	v7	**SL**	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
r9	v6	**SB**	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants
r8	v5		ARM-state variable-register 5.
r7	**v4**	**WR**	Variable register (v-register) 4. Thumb-state Work Register.
r6	**v3**		Variable register (v-register) 3.
r5	**v2**		Variable register (v-register) 2.
r4	**v1**		Variable register (v-register) 1.
r3	**a4**		Argument/result/scratch register 4.
r2	**a3**		Argument/result/ scratch register 3.
r1	**a2**		Argument/result/ scratch register 2.
r0	**a1**		Argument/result/ scratch register 1.

APCS Register Use Convention (2)

- Four argument registers which pass values into the function
- They must be saved across call if they contain values that are needed again
- They are **caller-saved register** variables when so used

Register	Synonym	Special	Role in the procedure call standard
r3	**a4**		Argument/result/scratch register 4.
r2	**a3**		Argument/result/ scratch register 3.
r1	**a2**		Argument/result/ scratch register 2.
r0	**a1**		Argument/result/ scratch register 1.

APCS Register Use Convention (3)

r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
r10	v7	**SL**	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
r9	v6	**SB**	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants
r8	v5		ARM-state variable-register 5.
r7	**v4**	**WR**	Variable register (v-register) 4. Thumb-state Work Register.
r6	**v3**		Variable register (v-register) 3.
r5	**v2**		Variable register (v-register) 2.
r4	**v1**		Variable register (v-register) 1.

- v1~v8, register variables which the function must return with unchanged values
- These are **callee-saved register** variables

APCS Register Use Convention (4)

Register	Synonym	Special	Role in the procedure call standard
r15		**PC**	The Program Counter.
r14		**LR**	The Link Register.
r13		**SP**	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
r10	v7	**SL**	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
r9	v6	**SB**	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants
r8	v5		ARM-state variable-register 5.
r7	**v4**	**WR**	Variable register (v-register) 4. Thumb-state Work Register.
r6	**v3**		Variable register (v-register) 3.
r5	**v2**		Variable register (v-register) 2.
r4	**v1**		Variable register (v-register) 1.
r3	**a4**		Argument/result/scratch register 4.
r2	**a3**		Argument/result/ scratch register 3.
r1	**a2**		Argument/result/ scratch register 2.
r0	**a1**		Argument/result/ scratch register 1.

Argument Passing

- The first 4 words arguments => a1 ~ a4
- Remaining words: push into the stack in reverse order
- **Floating point**

  (If floating-point values are passed through floating-point registers)

  – The first 4 floating-point arguments => f0~f3

  – All remaining arguments: the first 4 words => a1~a4

  – The remaining words => stack in reverse order

Effective Procedure Calls

- 四個或更少參數的函數比多於四個參數的函數執行效率要高
  - more than 4 arguments => use stack
- Caller
  - 減少對register / memory的存取動作
- Callee
  - 多了register可利用
- Inline function

Example

```
char* queue_bytes_v1(
  char* Q_start,
  char* Q_end,
  char* Q_ptr,
  char* data,
  unsigned int N)
{
  do {
     *(Q_ptr++) = *(data++);
     if (Q_ptr == Q_end)
        Q_ptr = Q_start;
  } while (--N);

  return Q_ptr;
}
```

```
typedef struct {
  char* Q_start,
  char* Q_end,
  char* Q_ptr
} Queue;
```

```
char* queue_bytes_v2(
  Queue* queue,
  char* data,
  unsigned int N)
{
  char* Q_start = queue->Q_start;
  char* Q_end = queue->Q_end;
  char* Q_ptr = queue->Q_ptr;
  do {
     *(Q_ptr++) = *(data++);
     if (Q_ptr == Q_end)
        Q_ptr = Q_start;
  } while (--N);

  return Q_ptr;
}
```

Result Return

- 1 word value in a1
- A value of length 2-4 words
  - a1-a2, a1-a3, a1-a4
- Indirect return (Memory)
  - Ex: return a structure with 8 words

Function Entry / Exit (1)

- A simple leaf function
  - Perform all its functions using only a1~a4
  - Have minimal calling overhead

```
            BL    leaf1
            ...
            ...

leaf1:      ...
            ...
            MOV   pc, lr   ; return
```

Function Entry / Exit (2)

- A general function

```
        BL      leaf2
        ...
        ...
leaf2:  STMFD   sp!, {regs, lr}    ; save registers
        ...
        ...
        LDMEA   sp!, {regs, pc}    ; restore and return
```

Backtrace

```
save code pointer        [fp]          fp points here
return link value        [fp, #-4]
return sp value          [fp, #-8]
return fp value          [fp, #-12]   points to next structure
[saved v7]
[saved v6]
[saved v5]
[saved v4]
[saved v3]
[saved v2]
[saved v1]
[saved a4]
[saved a3]
[saved a2]
[saved a1]
[saved f7]                             three words
[saved f6]                             three words
[saved f5]                             three words
[saved f4]                             three words
```

每個函式需儲存的資訊 (APCS)

The fp register points to the stack backtrace structure for the currently executing function.

crt0.s

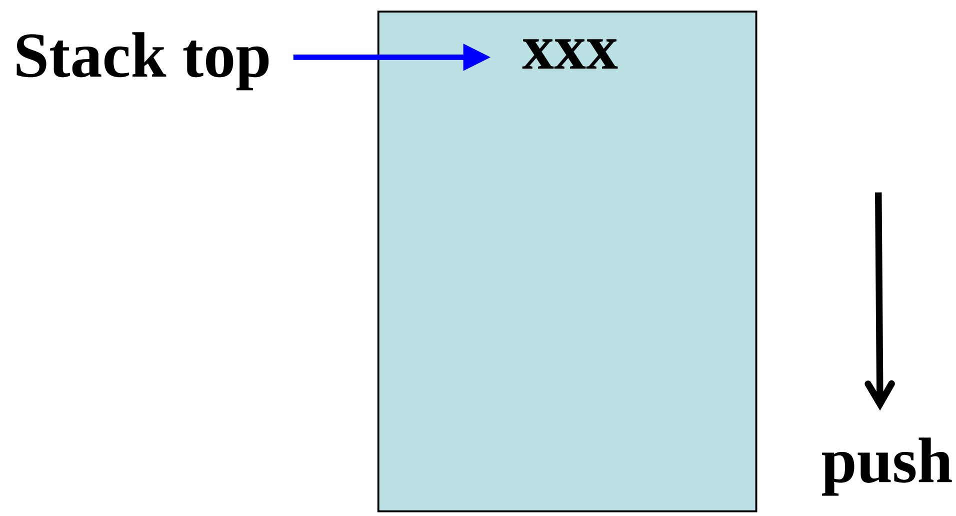
```
_mainCRTStartup:
 …
 mov r0, #0
 mov fp, r0
 …
 bl main
 …
```

```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

fp → 0

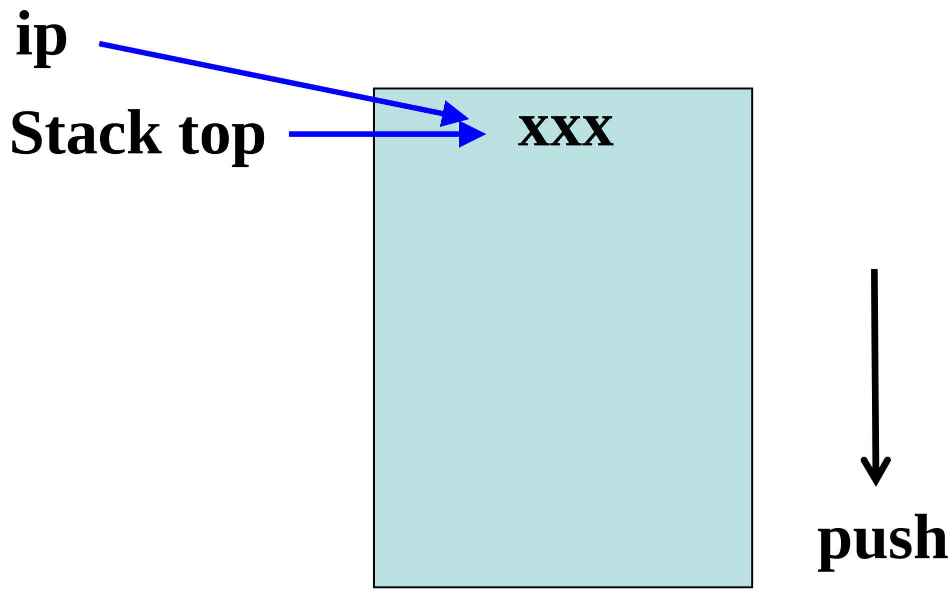
```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD  sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

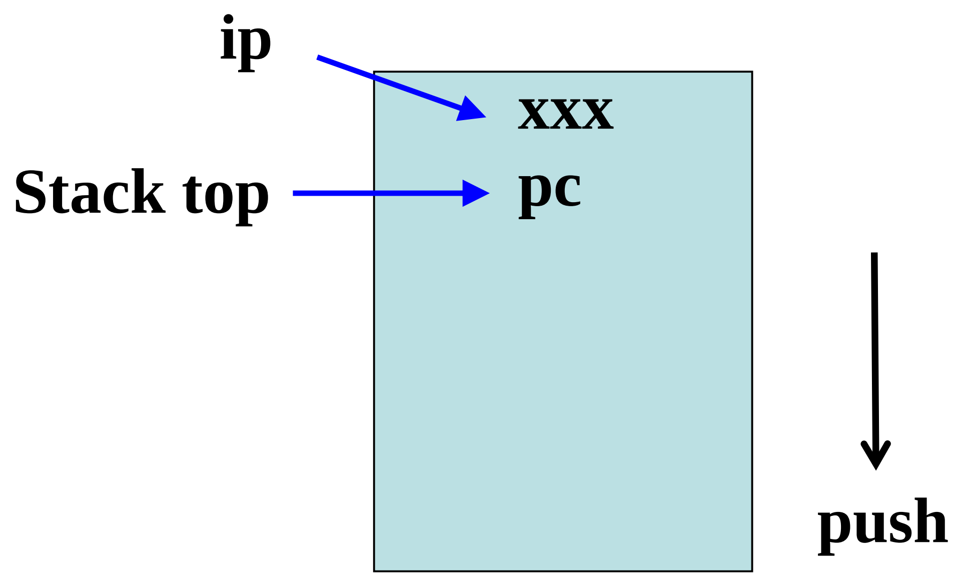
```
main:
   MOV    ip , sp
   STMFD  sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   ...
```



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```

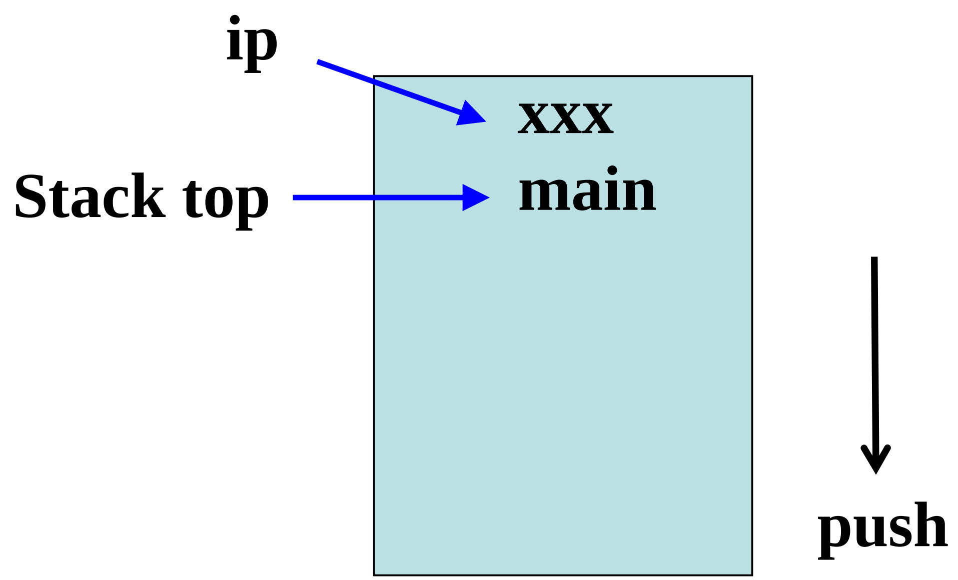
pc



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```

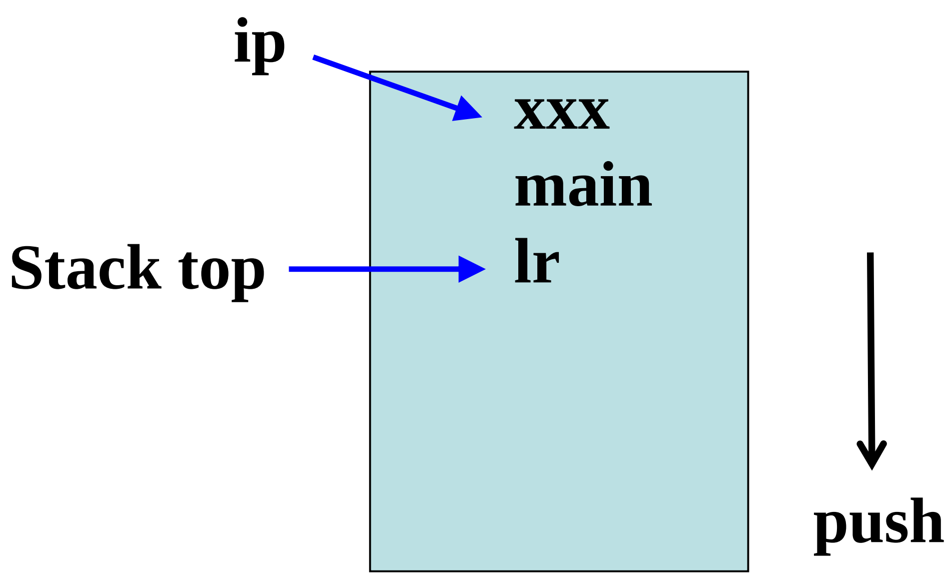
main



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD  sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```

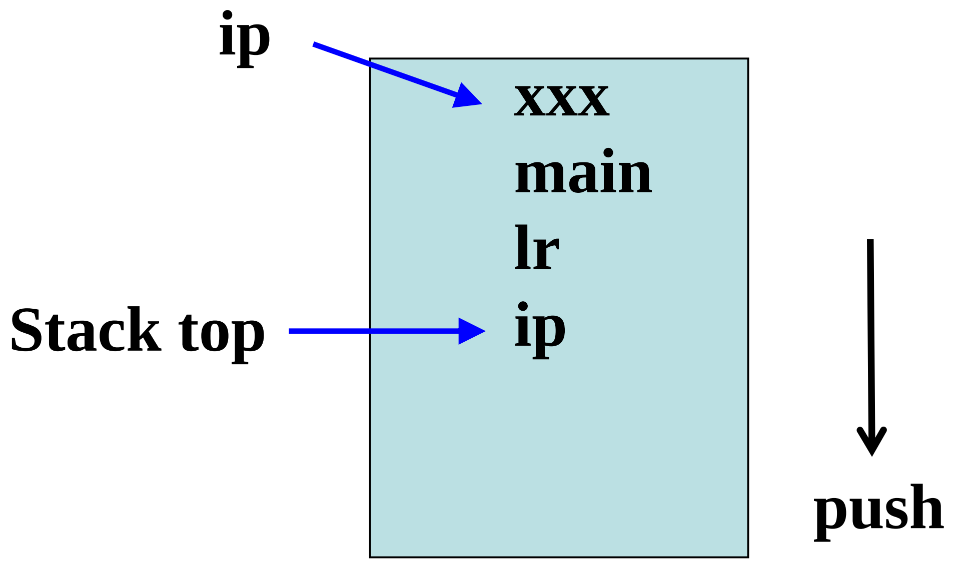
main



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD  sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```

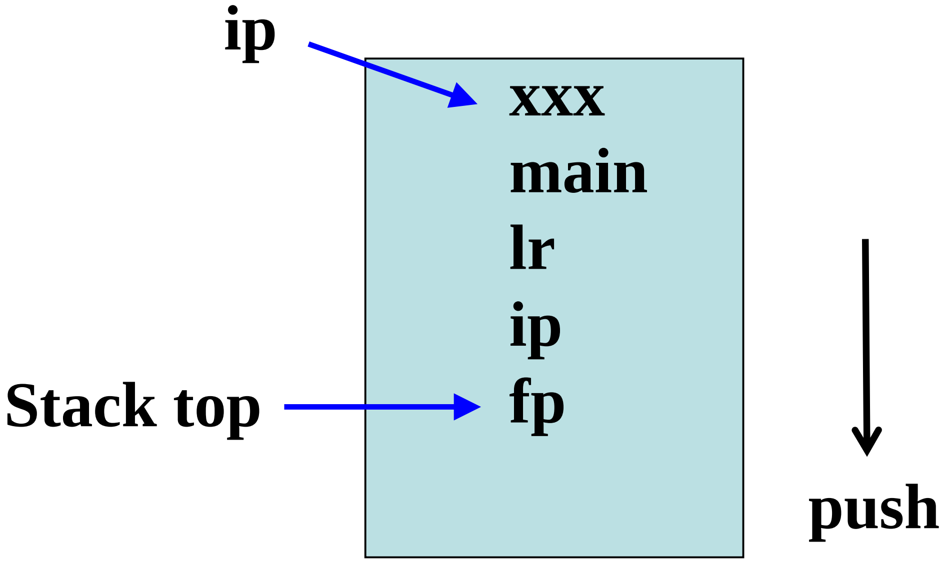
main



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD  sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```

main



```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

```
main:
   MOV    ip , sp
   STMFD sp!, {fp, ip, lr, pc}
   SUB    fp , ip, #4
   …
```

ip

xxx
main
return link
return sp
return fp → 0
...

```
func1()
{
   ...
}

main()
{
   ...
   ...
   func1()
   ...
}
```

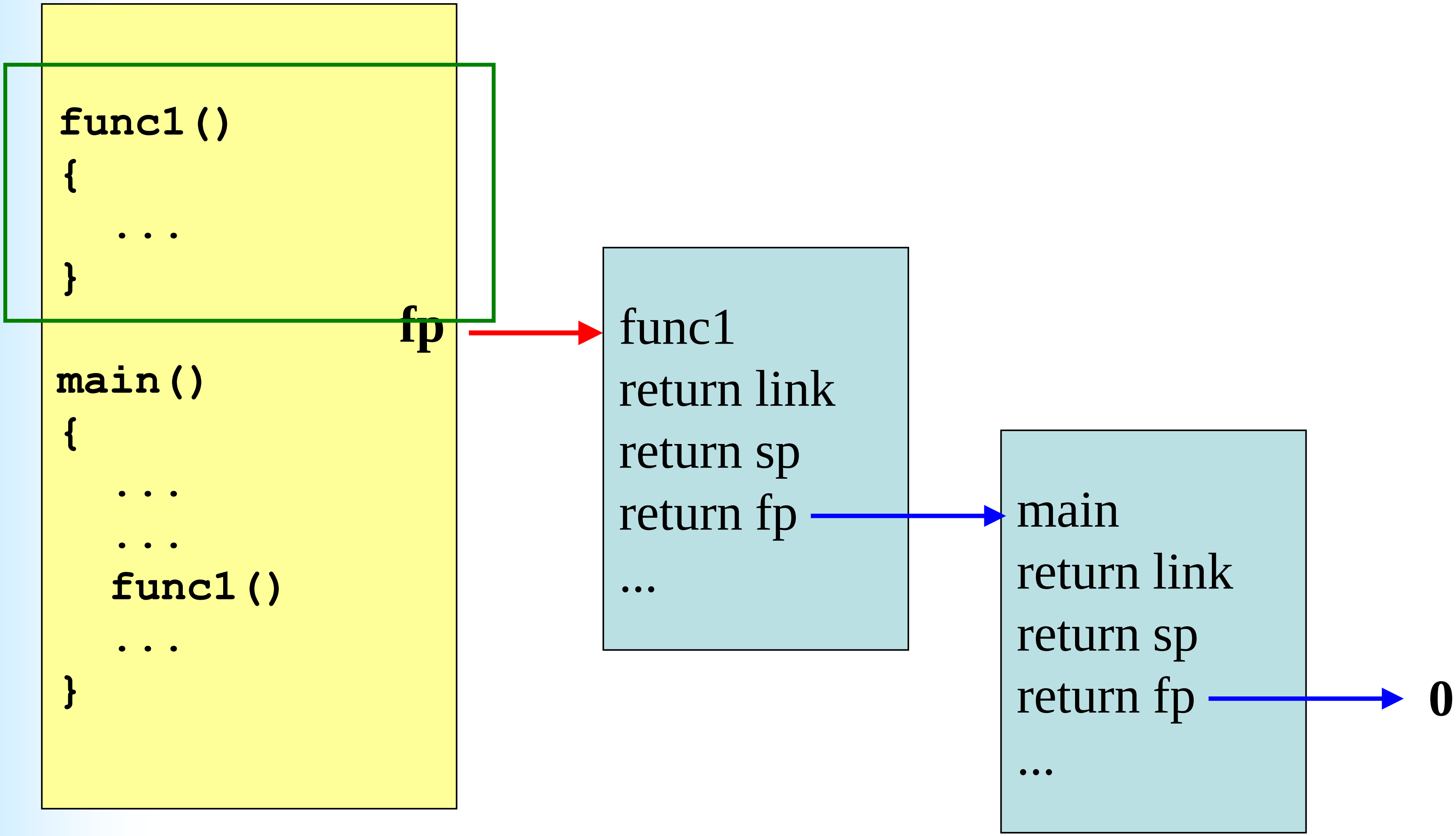
```
main:
   MOV     ip , sp
   STMFD   sp!, {fp, ip, lr, pc}
   SUB     fp , ip, #4
   …
```

pc

fp

main
return link
return sp
return fp
…

0


func1()
{
...
}
main()
{
...
...
func1()
...
}
fp
func1
return link
return sp
return fp
...
main
return link
return sp
return fp
...
0

Function Entry (APCS)

```
MOV ip, sp
STMFD sp!, {fp, ip, lr, pc}
SUB fp, ip, #4
```

OR

```
STMFD sp!, {r4-r10, fp, ip, lr, pc}
```

假如之後callee會用到r4-r10 register

Function Exit (APCS)

LDMEA fp, {fp, sp, pc}

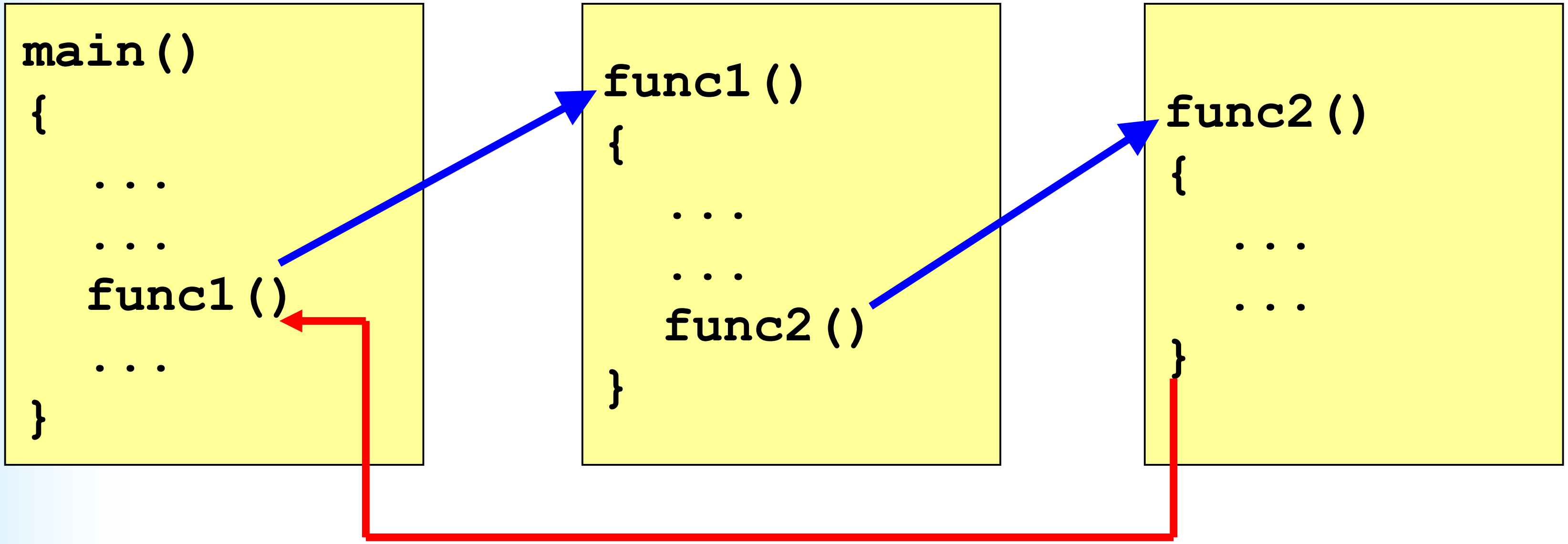
OR

LDMEA fp, {r4-r10, fp, sp, pc}

假如之前callee有先save r4-r10 register

Tail Continued Functions

- The compiler will cause the code to return directly from the continuing function



Inline Function

- Program will execute faster by eliminating the function-call overhead

```
void inc(int* b)
{
    (*b)++;
}

int main()
{
  int a = 10;

  inc(&a);
  ...
}
```

→

```
int main()
{
  int a = 10;

  (*(&a))++;
  ...
}
```

Inline Function in GCC

To declare a function inline, use the **inline** keyword in its declaration

```
inline void inc(int* b)
{
    (*b)++;
}

int main()
{
  int a = 10;

  inc(&a);
  ...
}
```

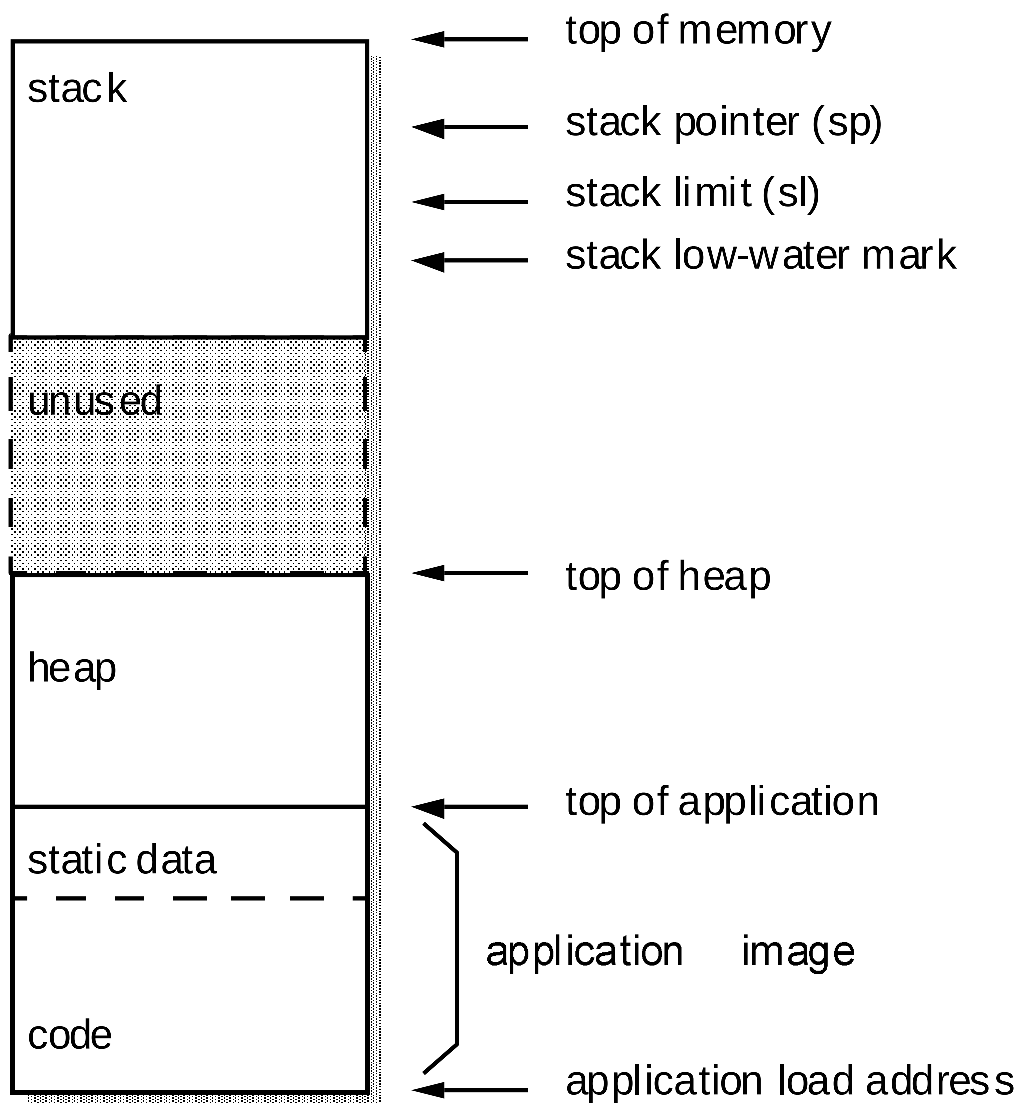
GCC does not inline any functions when not optimizing unless you specify the always_inline attribute for the function

```
inline void inc(int*) __attribute__((always_inline));
```

Outline

- Abstraction in software design
- Data types
- Floating-point data types
- Expressions
- Conditional statements
- Loops
- Functions and procedures
- **Use of Memory**
- Run-time environment

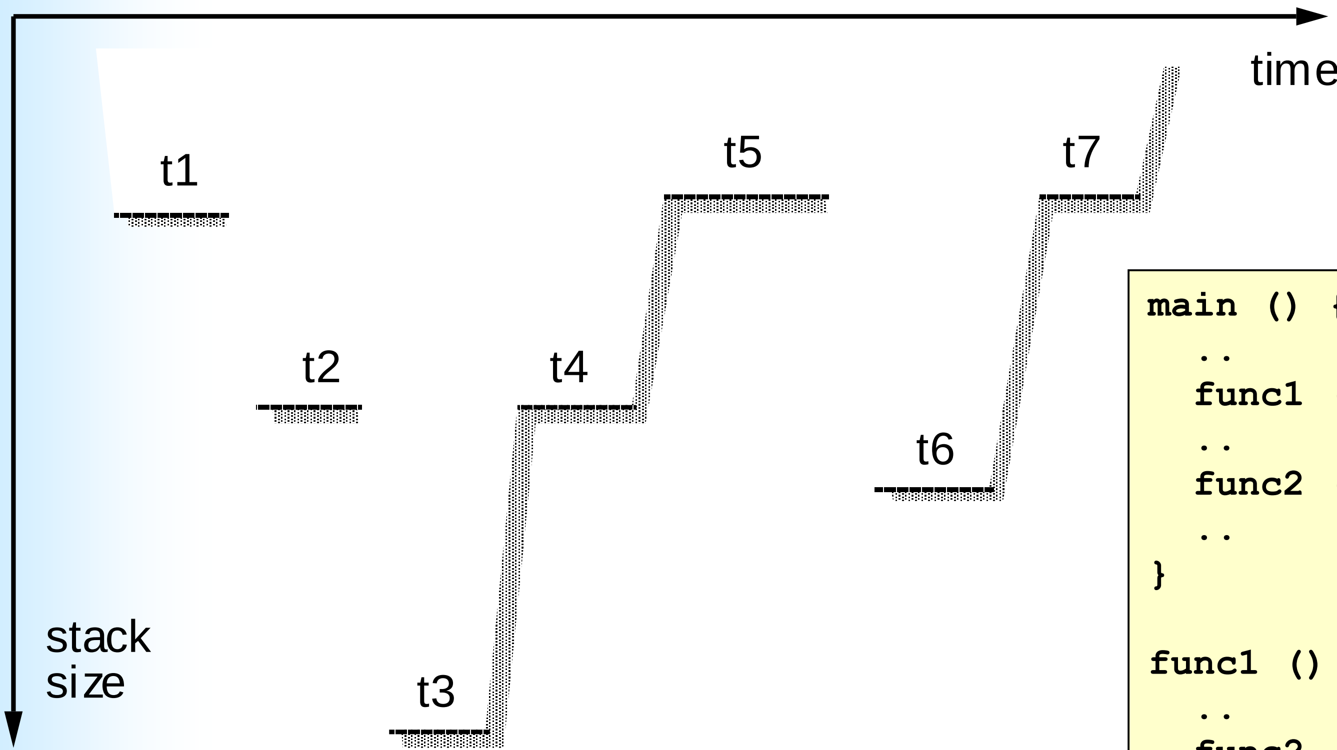
The Standard ARM C Program Address Space Model



A Simple Program

```
main () {
   ..                  /* t1 */
   func1 ();
   ..                  /* t5 */
   func2 ();
   ..                  /* t7 */
}

func1 () {
   ..                  /* t2 */
   func2 ();
   ..                  /* t4 */
}

func2 () {
   ..                  /* t3, t6 */
}
```

Stack Behavior



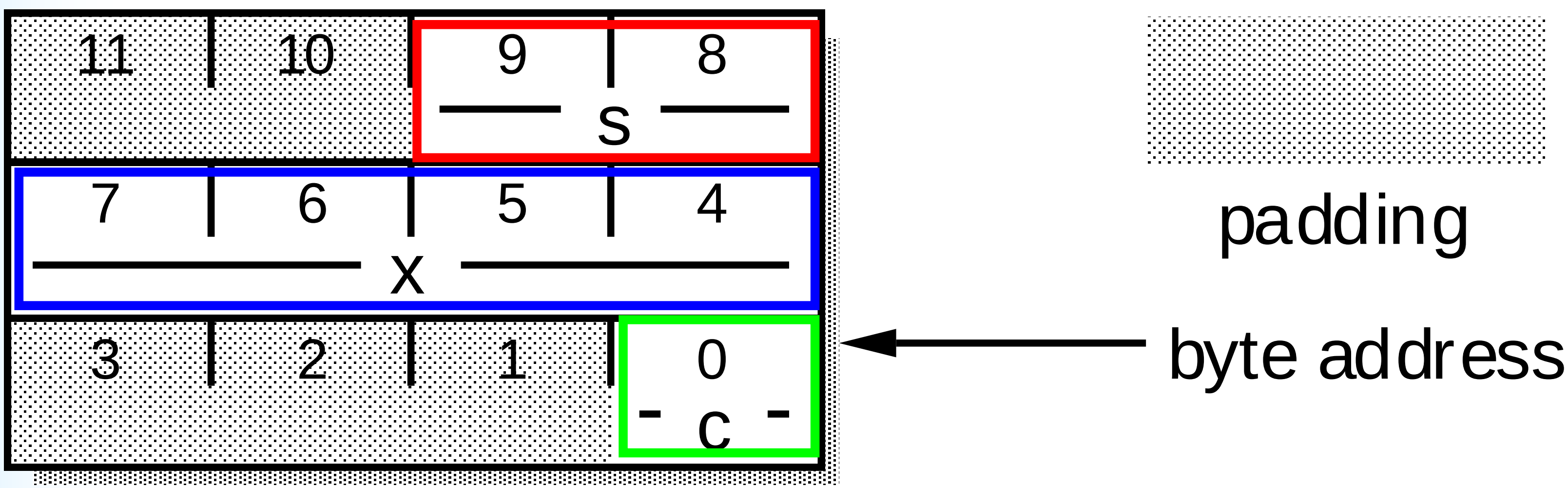

```
main () {
   ..                  /* t1 */
   func1 ();
   ..                  /* t5 */
   func2 ();
   ..                  /* t7 */
}

func1 () {
   ..                  /* t2 */
   func2 ();
   ..                  /* t4 */
}

func2 () {
   ..                  /* t3, t6 */
}
```

Memory Issues

- **Efficient : aligned data**
- **Inefficient: non-aligned data**
- ARM C compiler generally aligns data items on appropriate boundaries
  - Bytes are stored at any **byte** address
  - Half-words are stored at **even** byte addresses
  - Words are stored on **four-byte** boundaries

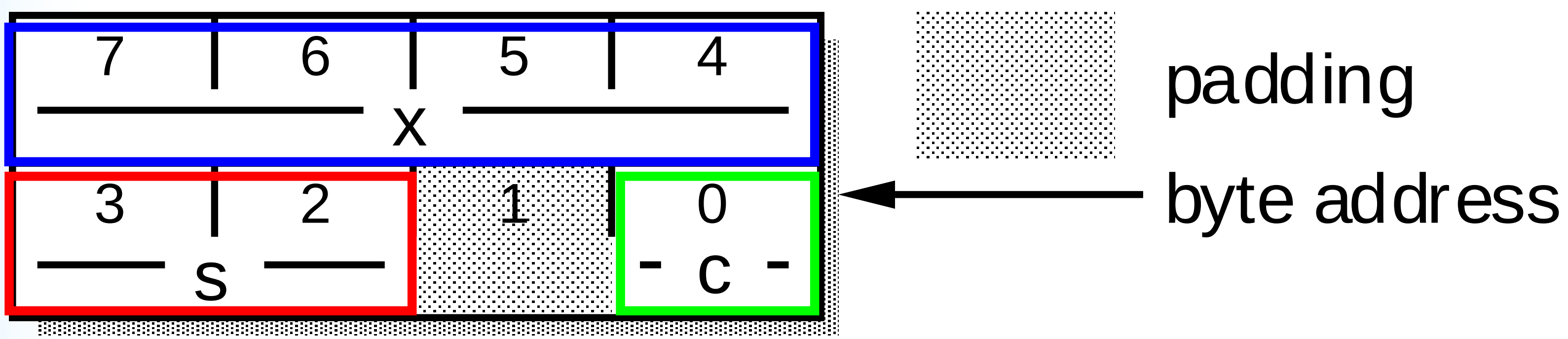
An Example:
Normal Structure Memory Allocation

```
struct S1 {
    char   c;
    int    x;
    short  s;
} example1;
```



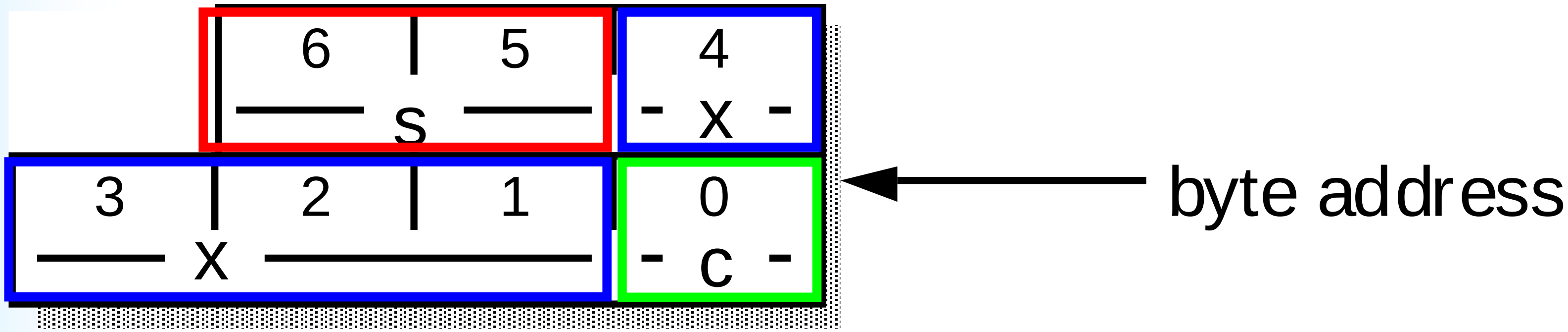
An Example: Efficient Structure Memory Allocation

```
struct S1 {
    char  c;
    short s;
    int   x;
} example1;
```



An Example:
Packed Structure Memory Allocation

```
__packed struct S1 {
    char   c;
    int    x;
    short  s;
} example1;
```



Variable Alignment in GCC (1)

- The keyword **__attribute__** allows you to specify special attributes of variables or structure fields
- This keyword is followed by an attribute specification inside double parentheses

Variable x is aligned on a 16-byte boundary

```
int x __attribute__ ((aligned (16)));
```

Variable Alignment in GCC (2)

```
struct S1 {
    char   c  __attribute__ ((align(1)));
    int    x  __attribute__ ((align(4)));
    short  s  __attribute__ ((align(2)));
} example1;
```

```
struct S1 {
    char   c  __attribute__ ((packed));
    int    x  __attribute__ ((packed));
    short  s  __attribute__ ((packed));
} example1;
```

Outline

- Abstraction in software design
- Data types
- Floating-point data types
- Expressions
- Conditional statements
- Loops
- Functions and procedures
- Use of Memory
- **Run-time environment**

Run-time Environment

- Software development
  - Compiler, Assembler, Linker, Debugger
  - **ANSI C Library**
    - File management
    - Input / Output
    - Real-time clock
    - …
- Embedded System
  - **Limited resources** (Cannot provide full ANSI C library)
  - Most of functions are irrelevant for different embedded systems
    - Depend on the function of the embedded system
    - Ex: Mobile phone, mp3 player, …etc.

Minimal Run-Time Library (1)

From ARM Limited: ~736 bytes

- **Division and remainder functions**
  – The ARM instructions set does not have divide instructions
- **Stack-limit checking functions**
  – A small embedded system has no memory management hardware
  – Ensure that programs operate safely
- **Stack and heap management**
  – C programs will use stack and heap during runtime

Minimal Run-Time Library (2)

- **Program start up**
  - The initialization of stack and heap, ex: crt0
- **Program termination**
  - Programs call _exit() when
    - Termination
    - an error is detected during runtime
  - _exit()
    - Flush all output streams, close all open streams
    - Remove all temporary files
    - … , finally, control is returned

Other Issues

- Fixed Point Arithmetic
- GCC inline assembly

Fixed Point: Idea (1)

$$1\ 2 + 3\ 5 = 4\ 7$$

$$1.2 + 3.5 = 4.7$$

Fixed Point: Idea (2)

1.2 + 3.5 = ?

1.2 + 3.5 = 4.7

Fixed Point: Idea

- 12 + 35 = 47
- **1.2 + 3.5 = 4.7**
- **似乎可以用整數指令來做浮點數的運算，只要小數點都點在固定的位置就可以了**
  - 假設register的值都需要把小數點點在第一位與第二位之間才是真正的數值
  - mov r1, #12
  - mov r2, #35
  - add r2, r1, r2

雖然r2的值是47，但是我們解讀為4.7

Fixed Point Arithmetic (1)

- Floating point
  - IEEE-754
  - Fixed point
- A pair of integers (n, e) represents the fraction
  - n: mantissa
  - e: exponent

$$\text{Fraction} = n \times 2^{-e}$$

Fixed Point Arithmetic (2)

Mantissa (*n*)	Exponent (*e*)	Binary	Decimal
01100100	-1	011001000.	200
01100100	0	01100100.	100
01100100	1	0110010.0	50
01100100	2	011001.00	25
01100100	3	01100.100	12.5
01100100	7	0.1100100	0.78125

- If e is known at compile time, (n, e) is said to a fixed point number
- Fixed point numbers can be stored in standard integer variables by storing the mantissa

Fixed Point Arithmetic (3)

- The exponent e is usually denoted by the letter q
- Ex: q=14, 0x00004000 represents ?

000000000000000001|00000000000000

$$F = 0x00004000 \times 2^{-14} = 1$$

Examples

- Ex: q=14, 0x00000001 represents ?

$$F = 0x00000001 \times 2^{-14} = 2^{-14}$$

Change of Exponent

- Change the exponent from p to r

$$\text{Fraction} = n \times 2^{-p} = \boxed{(n \times 2^{r-p})} \times 2^{-r}$$

- Mantissa = n << (r-p) if (r >= p)

  n >> (p-r) if (p > r)

小數點對齊，才可以直接做運算

Shift operation

Addition and Subtraction

- Operation: c = a + b
- Convert a and b to have the same exponent as c

$$a+b=n\times 2^{-r}+m\times 2^{-r}=(n+m)\times 2^{-r}=c$$

```
; a is in register r0
; b is in register r1
; a, b and c have the same exponent

ADD r2, r0, r1
```

Example

- $3.7 + 1.21 = ?$
- $37 * 10^{-1} + 121 * 10^{-2}$
- $(37 * 10) * 10^{-2} + 121 * 10^{-2}$
- $370 * 10^{-2} + 121 * 10^{-2}$
- $491 * 10^{-2}$

- $3.7 + 1.21 = 4.91$

Fixed Point Arithmetic (4)

- If the processor does not support floating-point operations
  - Do floating-point operations by software
    - Software emulation (IEEE-754)
    - Fixed point
  - Fixed point computation is faster than software emulation (IEEE-754), but less accuracy, informal.

Inline Assembly

GNU Inline Assembly (1)

“asm” and “__asm__” are valid

```
asm(“add r2, r1, r0”);
__asm__(“add r2, r1, r0”);
```

GNU Inline Assembly (2)

- “\n” => newline
- “\t” => tab

```
__asm__("add r2, r1, r0\n\t"
        "mov r3, r2\n\t"
        "mul r0, r1, r3");
```

Example

```
int main(void)
{
    int a;

    a = 100;
    __asm__("add r2, r1, r0");
    printf("%d\n", a);

    return 0;
}
```

useless

GNU Inline Assembly (3)

- Basic format

```
asm ( assembler template
        : output operands               /* optional */
        : input operands                /* optional */
        : list of clobbered registers   /* optional */
);
```

GNU Inline Assembly (4)

```
int a=10, b;                a               b
asm ("mov r1, %1;  mov %0, r1;"
        :"=r"(b)      /* output */
        :"r"(a)       /* input */
        :“r1”         /* clobbered register */
);
```

Tell GCC that the value of r1 is to be modified inside "asm", so GCC won’t use this register to store any other value

```
int main(void)
{
    int m=2010, n=1, k=6, p=1010;
    asm ("sub r2,%1,#10;
          add r2,r2,%3;
          add r2,r2,%2;
          mov %0,r2"
          :"=r"(p)
          :"r"(m),"r"(n),"r"(k)
          :"r2"      /* clobbered register */
    );
    printf("%d %d %d %d\n", m, n, k, p);
    return 0;
}
```

Backup

Conditional Statements: switch…case (4)

- 如果switch發生的條件是大範圍的x
- 利用hash function