C Programming Course

RSO/Microcomputer Tool Marketing Department

Yugo Kashiwagi

The Objective of This Course

- To understand techniques for Embedded Programming
  - Difference from PC Programs
  - How to control processor using C Language
- To acquire basic knowledge to be a Professional Programmer
  - Total productivity including design, coding, testing, maintainance
  - Team development

Prerequisits

- Basic knowledge of C (or C++, Java, C#) language
- Basic knowledge of CPU
  - Processor, memory, peripherals
  - Registers, Instructions

Contents

I. Quick Review of the C Language

Contents

1. Introduction

- The Purpose of Programming Languages
  - Give instruction to a computer to do some job.
  - Express ideas and communicate with other programmers (including yourself!).

History of Programming Languages

- '60s
  - Languages for Scientific/Business processing
    - FORTRAN, ALGOL/Scientific, COBOL/Buisiness
- '70s-'80s
  - System Programming Languages
    - Pascal, C
- '90s and later
  - Descendents of C
    - C++, Java, C#

History of C

- 1972: Designed to write UNIX operating system
- 1978: Kernighan & Ritchie
  "The C Programming Language"
- 1990: ISO Standardization
- 1999: C99 ISO Standardization

References

- Kernighan & Ritchie
  "The C Programming Language"
  (Textbook by the inventors, The Bible)
- ISO/IEC 9899-1990: Programming Languages - C (Standard)

How do we Review the C Language?

- Quick review of C grammar based on programs from introductory course (you should know them already)
- Advanced topics are added
  - Portability consideration
  - Implementation issues
  - Higher programming techniques

Ex. 1.1: A Simple Program

```
#include <stdio.h>

main()    /* First Example */
{
    int a;
    a=10;
    printf("value of a: %d\n", a);
}
```

Components of Program

```
#include <stdio.h>

main()    /* First Example */
{
    int a;
    a=10;
    printf("value of a: %d\n", a);
}
```

Directive

Comment

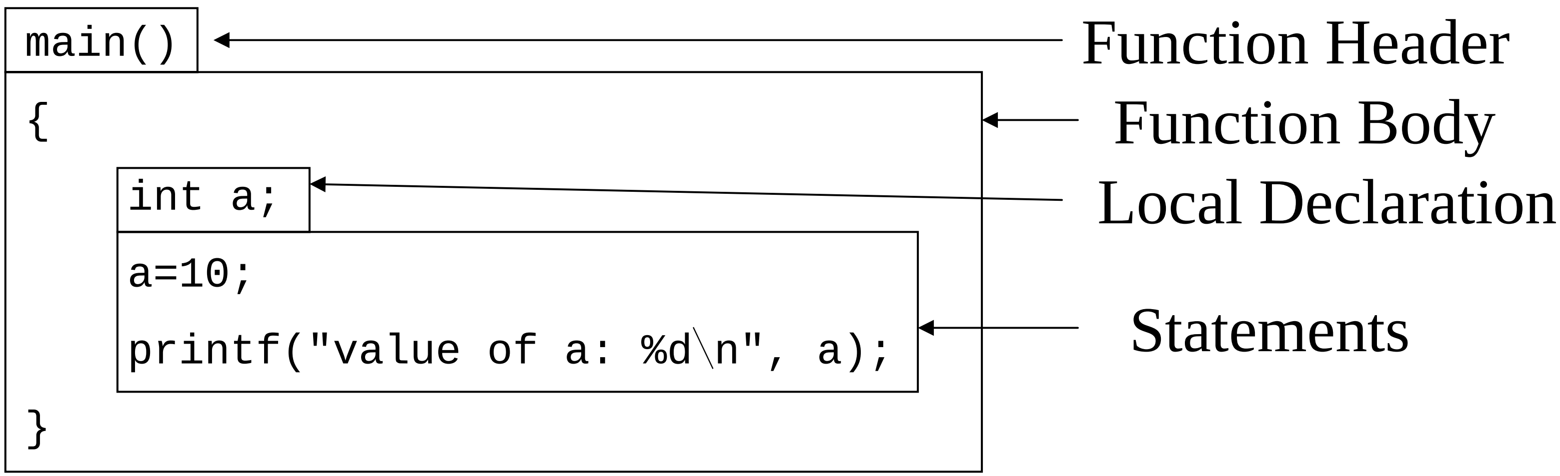
Function Declaration

- Directives starts with #, and specifies command to the compiler.
- A function declaration defines a data handling procedure.
- Comment is an annotation and has nothing to do with the program execution. Enclosed by `/*` and `*/`.
- The program starts execution from the function `main`.

Elements of Program

- Keywords: `int`, etc.
  - Have a fixed meaning.
  - Cannot be used as an identifier.
- Identifiers: `main`, `a`, `printf`, etc.
  - Name data or a functions
- Literals: `10`, `"% value of a: %d\n"`, etc.
  - Represent constant data
- Operators, separators: `(`, `)`, `=`, `;`, `{`, `}`, etc.

Structure of a function



- Function header defines function name and its interface.
- Function body, enclosed by { and } defines the processing of the function.
  - Local declaration defines data used in a function.
  - Statements describe actual data processing

Data Types

- Every C data (constant or variable) has a type.
- Every variable must be given a type by a declaration, before it is used.

$\boxed{\texttt{int}}\ \boxed{\texttt{a}}\ \texttt{;}$

Specifies data type

Specifies the name of the variable

Basic Data Types

- `short` (16 bit), `int` (16/32 bit), `long` (32 bit)
  - `int` size depends on processor
  - Signed data
- `unsigned short`, `unsigned`, `unsigned long`
  - Unsigned data
- `float` (32 bit), `double` (64 bit)
  - Floating point data
- `char` (8 bit), `unsigned char`
  - ASCII-code or 8-bit integer)

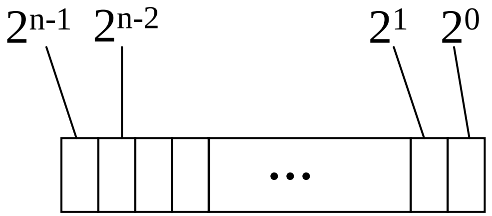
Size of `int`

- For SH, size of `int` is 32 bit (size of registers).
- <u>If data size should be explicit, do not use `int`. Use `long` or `short`.</u>
- <u>Use `int` if you are sure the data value is small enough (e.g. index of an array) for compatibility.</u>
- Use of `short` can be inefficient on SH (or other 32-bit machines).

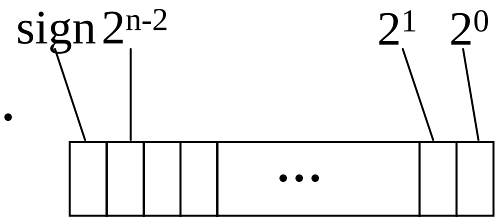
Why `short` is Not Efficient on 32-bit Machines

- `short` data must be in the range of short ($-2^{15} \sim 2^{15}-1$ for signed, $0 \sim 2^{16}-1$ for unsigned)
- So after each operation, the instruction to keep the data value in this range must be executed (in SH, `EXTS.W` for signed, `EXTU.W` for unsigned).
- Similar for `char` type.

Data Representation



- n-bit unsigned integer represents $0\sim2^n-1$.



- n-bit signed integer represents $-2^{n-1}\sim2^{n-1}-1$.
  - The MSB is interpreted as -2^{n-1}.
    e.g. 111....11 is -1 $(-2^{n-1}+(2^{n-2}+2^{n-3}+...+1))=-1$
  - Two's complement representation.
- Floating point data is represented by IEEE 754 Floating Point Standard Format
  - Float: 1 bit sign, 8 bit exponent, 23 bit mantissa
  - Double: 1 bit sign, 11 bit exponent, 52 bit mantissa

`#include` directive

- `#include` make the compiler to include another file.
- `<stdio.h>` is the file which contains necessary definitions to use standard I/O functions.
  (They are just C source files, nothing special)
- `<>` encloses include files provided by the system ("/usr/include" in Unix systems)
- `""` encloses user-defined include files

Strings

- Strings (enclosed by `""`) defines character string data (an array of character)
- Terminating null character is automatically appended. So the string "abc" occupies 4 (=3+1) bytes.
- `"\"` introduces an escape sequence
  - `\n` newline
  - `\t` tab
  - `\\` \ itself
  - `\"` "
  - `\0` Null character (ASCII code 0)

`printf` function

- `printf` is a formatted output function
  - Declared in `<stdio.h>`.
  - First parameter specifies output format.
  - Format is specified by the character (e.g. `%d`: decimal output).
  - Rest of the parameters are converted/output according to the format specification.
  - `printf` is available only if the system supports I/O library (not usually available in embedded systems).
- `printf` accepts variable number of arguments. To implement such a function, you need to use a library `<vaarg.h>` (not covered in this lecture).

2. Expressions

- Expressions computes values.
- C has a rich set of operators.
- Even an assignment is an operator in C.
- Some expressions have side effect (e.g. assignment expression).

Ex. 2.1: Arithmetic Operators

```
#include <stdio.h>

main()
{
    int a, b;
    double c, d;
    a=5;
    b=3;
    printf("%d %d %d %d %d %d\n", -a, a+b, a-b, a*b, a/b, a%b);
    c=1.0;
    d=2.0;
    printf("%f %f %f %f %f\n", -c, c+d, c-d, c*d, c/d);
}
```

Arithmetic Operators

- `+`, `-` (unary and binary), / has its usual meaning in mathematics.
- `*`: multiplication, `%`: remainder
- Operator precedence
  - Unary operators has highest precedence.
  - Multiplicative operators (`*`, `/`, `%`) has higher precedence than additive operators (`+`, `-`).
  - Same level operators associates to the left.
- `/` and `%` operators for negative integers are not well-defined. So use them only for positive integers.

Ex. 2.2: Logical Operators

```
#include <stdio.h>

main()
{
    unsigned int a, b, c;
    a=0x000000ff;
    b=0x00000f0f;
    c=3;
    printf("%x %x %x %x %x %x\n",
           ~a, a&b, a|b, a^b, a<<c, a>>c);
}
```

Logical Operators

- &: and, `|`: or, ^: exclusive-or, ~: negation (unary), <<: left shift, >>: right shift
- Use logical operators with unsigned data. (because their effect on sign-bit is not well-defined).
- Hexadecimal notation (hexadecimal digits starting with `0x`) is convenient to represent bit patterns.
- `%x` in format string specifies hexadecimal output (without `0x`).

Operator Precedence (1)

- (Left/Right) shows associativity of the operator.
- Postfix operators: `( )`, `[ ]`, `++`, `--`   (Highest)
- Unary operators: `+`, `-`, `*`, `&`, `!`, `-`
- Multiplicative operators (Left): `*`, `/`, `%`
- Additive operators (Left): `+`, `-`
- Shift operators (Left): `>>`, `<<`
- Relational operators (Left): `>`, `<`, `>=`, `<=`
- Equational operators (Left): `==`, `!=`

Operator Precedence (2)

- Bitwise and operator (Left): &
- Bitwise exclusive or operator (Left): ^
- Bitwise or operator (Left): |
- Logical and operator (Left): &&
- Logical or operator (Left): ||
- Conditional operator: ? :
- Assignment operator (Right): =, +=, -=, etc.
- Comma operator (Left): , (Lowest)

Precedence and Associativity of Operators

- Higher precedence operator is applied first
  - e.g. `a*b+c` is interpreted as `(a*b)+c`
- In case of same precedence, associativity specifies the operator to be applied first
  - e.g. `a/b*c` is interpreted as `(a/b)*c` (left associative)
  - e.g. `a=b=c` is interpreted as `a=(b=c)` (right associative)
- <u>When you are not sure, use parentheses.</u>

Arithmetic Promotion

- `char`, `short`, `unsigned char`, `unsigned short` types are first converted to `int` type.
- Other types are promoted according to the ordering of the next page to convert both operands into the same type.
- <u>If you are not sure, use cast (explicit type conversion) to avoid mixed-type operations.</u>

Arithmetic Promotion Rules

When `int` is 32 bits

$$
\begin{tikzcd}
 & \text{double} & \\
 & \text{float} \arrow[u] & \\
 & \text{unsigned long} \arrow[u] & \\
\text{long} \arrow[ur] & & \text{unsigned} \arrow[ul] \\
 & \text{int} \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

When `int` is 16 bits

$$
\begin{tikzcd}
\text{double} \\
\text{float} \arrow[u] \\
\text{unsigned long} \arrow[u] \\
\text{long} \arrow[u] \\
\text{unsigned} \arrow[u] \\
\text{int} \arrow[u]
\end{tikzcd}
$$

You don't have to remember this.
Make sure you don't use mixed-type arithmetic.

Arithmetic Overflow

- Unsigned arithmetic is computed modulo 2^{32}. No overflow occurs (guaranteed by the standard)
  - e.g. 0xffffffff+0xffffffff=0xfffffffe
- <u>In signed arithmetic, the result is not guaranteed by the standard (but usual implementation computes modulo 2^{32}).</u>
  - e.g. 2147483647+2147483647 -> overflow

Notes on Arithmetic

- Don't mix different types in arithmetic (types with different size, signed/unsigned)
- Assume that signed arithmetic overflow is not guaranteed.
- Don't apply logical operators to signed data.
- >> is not defined for negative signed integers in standard.
- There is a case when signed division overflows (namely, (-2147483647)-1)/(-1)

Notes on Side Effects

- The side effect occurs in assignment/increment/decrement/function call operations.
- The order of side effects is not specified in a statement.
- <u>Don't use more than one operations with side effects in one statement.</u>
  - e.g. `a[i++]=b[i++];`

3. Statements

- Basic statements are expressions (expressions include assignment and function calls)
- There are several ways to build more complex statements:
  - Selection: `if` and `switch` statements.
  - Repetition (loop): `for` and `while` statements.

Ex. 3.1: `if` statements

```
#include <stdio.h>

int max(int a, int b, int c)
{
    int maxval;
    if (a>b)
        maxval=a;
    else
        maxval=b;
    if (c>maxval)
        maxval=c;
    return maxval;
}

main()
{
    printf("%d\n", max(3, 4, 5));
}
```

Function Interface

Type of the function

Function parameters

```
int max(int a, int b, int c)
{
    ...
    return maxval;
}
```

Return statement

- Type of the function is the type of its return value.
- Function parameters give the declaration of the function input.
- The return statement specifies the return value of the function, and the value must be compatible with the type of the function.
- <u>Don't forget to return a value, otherwise you receive indeterminate data.</u>

`if` Statement

- The condition (inside `()`) is evaluated, and one of two statements is evaluated.
- If else `part` does not exist, no processing is done if the condition fails.

Conditions

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

```
if (c>maxval)
    maxval=c;
```

Executed when condition holds

Executed when condition fails

Compound Statements

- A sequence of statements can be handled as one statement by enclosing them by `{}`.

The range of `if` statement

```
if (c>maxval)
    maxval=c;
    maxval1=c1;
```

```
if (c>maxval){
    maxval=c;
    maxval1=c1;
}
```

This statement is not dominated by `if`, and always executed (indentation is not the part of syntax).

Combination of `if` statements

- `if` statement itself can be a part of another `if` statement.
- Following construct is useful for multi-way selection.

```
if (a==0)
  zero();
else if (a==1)
       one();
     else
       others();
```

Nested `if` statement

Data Type of Conditions

- There is no special data type for conditions. They are just `int`.
- 0 represents false, and all the other data represents true.
- Relational, equational, and other logical operators (`!`, &&, `||`) returns 0 for false, and 1 for true.

Ex. 3.2: `switch` statements

```
#include <stdio.h>

main()
{
    int c;
    c=getchar();
    switch (c){
    case 'Y':
        printf("OK\n");
        break;
    case 'N':
        printf("NG\n");
        break;
    default:
        printf("Illegal Character\n");
        break;
    }
}
```

Character literal

- Character literal specifies one-character data.
- Character literal represents an integer constant whose value is its ASCII code.
- Its syntax is similar to strings, but enclosed by `''`.
  - e.g. `'Y'`, `'N'`.
- The character `'` itself is escaped as `'\''`.

`getchar` function

- `getchar` function inputs one character from standard input.
- It is declared in `<stdio.h>` like `printf`.

`switch` Statement

- Switch statement selects according to the value.

```
switch (c){
  case 'Y':
    printf("OK\n");
    break;
  case 'N':
    printf("NG\n");
    break;
  default:
    printf("Illegal Character\n");
    break;
}
```

Value to be tested

`case` labels specify where to start the execution

`break` statement exits the `switch` statement

`default` means that no case label matches

`switch` Statement

- `switch` statement evaluates the value, and starts execution from the matching `case` label in the compound statement.
- When no case label matches, the execution starts with `default` label. If `default` label does not exist, no processing is done.
- `switch` statements ends when the execution reaches end of the compound statement, or it encounters `break` statement.

Notes on `switch` Statement

- More than one case label can be specified in the same place.

```
switch (c){
case 0: case 1: case 2:
    small();
    break;
case 3: case 4: case 5:
    large();
    break;
default:
    break;
}
```

Notes on `switch` Statement

- If you don't write `break` statement, the execution fall through the `case` label.

```
switch (c){
case 0:
    a=1;
case 1:
    a=2;
    break;
default:
    break;
}
```

When `c=0`, the execution falls through the label `case 1:`, and the final value of `a` will be 2.

Such a program is difficult to understand. <u>Write `break` statement corresponding to each case label.</u>

Notes on `switch` Statement

- <u>Don't omit the `default` label, even if there is nothing to do in the default case.</u>
- It is a good idea to check errors for unexpected cases.

Ex. 3.3: `for` statements

```
#include <stdio.h>

main()
{
    int i;
    for (i=0; i<10; i++)
        printf("%d %d %d\n", i, i*i, i*i*i);
}
```

`for` Statement

- for statement specifies a loop with initialization, termination condition, and what to do at the end of each step.

The loop continues while this condition holds.

Initialization

Step processing, executed after each step of the loop.

```
for ( i=0 ; i<10 ; i++ )
printf("%d %d %d\n", i, i*i, i*i*i);
```

`i++` is increment operator, and has the same effect as `i=i+1`.

Loop body, repeated while the condition holds.

Ex. 3.4: while statements

```
#include <stdio.h>

main()
{
    int c;
    while ((c=getchar())!=EOF){
        if (c>='a' && c<='z')
            c=c-'a'+'A';
        putchar(c);
    }
}
```

`while` Statement

- while statement specifies a loop with a termination condition.

Loop continues while this conditino holds.

```
while ((c=getchar())!=EOF){
   if (c>='a' && c<='z')
       c=c-'a'+'A';
   putchar(c);
}
```

Body of the loop

`break` statement

- `break` statement is also used to exit `for`/`while` loops.

```
for (i=0; i<100; i++){
  if (a[i]==0)
    break;          exits the loop
  ...
}
```

More on `<stdio.h>`

- `putchar` is a function which outputs a character to standard output.
- The type of `getchar` is `int` (not `char`) to include `EOF` special value for end-of-file case other than normal `char` data as a return value (note that `c` is declared as `int`.)
- Both `putchar` and `EOF` is defined in `<stdio.h>`

Values of Assignment Expression

- `(c=getchar())==EOF` compares the value of assignment with a constant `EOF`.
- In C, assignments have values, and its value is the assigned value.

Another Way to Write the Example Program

- Here is another way to write the example loop (may be easier to understand).
- In earlier C versions, the former is preferred because of more compact code, but now, better compiler optimization and more memory encourages the latter code, which is easier to understand.

```
c=getchar();
while (c!=EOF){
    if (c>='a' && c<='z')
        c=c-'a'+'A';
    putchar(c);
    c=getchar();
}
```

Character Constant as Integer Data

- Remember that character constants are just integer constant with the value of ASCII code of the character.
- `c-'a'+'A'` converts lower case letters to upper case letters.

Other Statements

- `do` <statement> `while (`<condition>`);`
  - Repeats <statement> while <condition> holds.
  - <statement> is executed at least once.
- `continue;`
  - Restarts the loop (goes back to the beginning of iteration).
- `goto` <label>`;`
  - Jumps to the program point specified by <label>
  - <u>Don't use `goto` statements.</u>

When `goto` is Appropriate

```
  while (i<100){
    while (j<100){
      ...
      if (error)
        goto loop_exit;
    }
  }
loop_exit:
```

You cannot exit from a nested loop with single `break` statement.

4. Data Structures

- Arrays define a homogeneous (same type) set of data.
- Structures define a heterogeneous (different type) set of data.
- Unions define a set of data sharing the same memory location.
- Pointers allow links between data.

Ex. 4.1: Arrays (1)

```
#include <stdio.h>

char buf[81];

void getline(void)
{
    int i, c;
    i=0;
    while ((c=getchar())!='\n'){
        buf[i]=c;
        i++;
    }
    buf[i]='\0';
}
/* to be continued */
```

Ex. 4.1: Arrays (2)

```
/* continued */
main()
{
    int i;
    getline();
    i=0;
    while (buf[i]!='\0'){
        putchar(buf[i]);
        i++;
    }
    putchar('\n');
}
```

Arrays

- `char buf[81]` declares an array of `char` whose number of elements (array size) is 81.
- Array index is 0-based, i.e. the first member is `buf[0]` and the last member is `buf[80]`.
- An array element is referenced as `buf[i]`, where `i` is the index of the array.
- <u>Make sure that array index is within the array range.</u>

Memory Allocation of an Array

- Array elements are allocated consecutively in memory.

```
int a[81];
```

a[0]	a[1]	a[2]	...	a[80]

Multidimentional Arrays

- `int a[5][10]` declares an two-dimentional (5*10) array of `int`.
- Multidimentional array is referenced like `a[i][j]`.
- This declaration is interpreted as "array of 5 (array of 10 `int`)".

Multidimentional Array Allocation

- `int a[5][10]` is allocated in the following way.
- The last index changes the fastest (Row Major)

a[0][0]	a[0][1]	a[0][2]	...	a[0][9]
a[1][0]	a[1][1]	a[1][2]	...	a[1][9]
:				
:				
a[4][0]	a[4][1]	a[4][2]	...	a[4][9]

`void` type

- The type `void` is no type. It is used to indicate that the function receives no parameters or returns no value.
- Without type specification, the function returns `int` (`main` function returns the value to return to operating system).
- Use `void` to indicate no parameters/no return value explicitly.

Global Data

- Declarations at the top level declares global data (e.g. `char buf[81];`)
- Global data can be used from every function after its declaration.
- On the other hand, declaration in a function is visible only inside the function.

Ex. 4.2: Structures

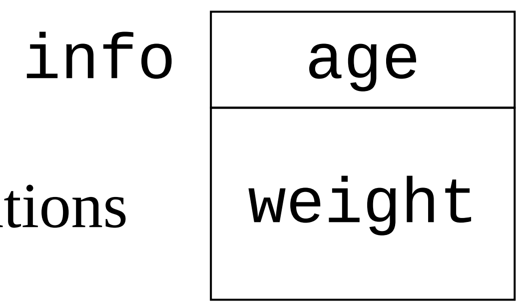
```
#include <stdio.h>
struct{
    int age;
    double weight;
} info;

main()
{
    int a;
    double w;
    scanf("%d", &a);
    while (a>=0){
        scanf("%lf", &w);
        info.age=a;
        info.weight=w;
        printf("age: %d weight: %f\n", info.age, info.weight);
        scanf("%d", &a);
    }
}
```

Structures

- Structure declares a type with a set of component members (`struct { ... }` specifies a type, just like `int`).
- The struct members are allocated consecutively (except gap).
- Struct members are referenced like `info.age`, `info.weight`, etc.

```
struct{
    int age;
    double weight;
} info;
```

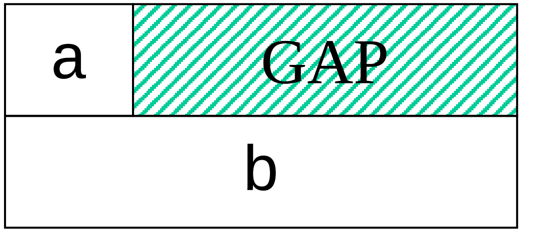
Member declarations



Data Alignment

- Some data types should be aligned.
- For SH, `short` must be aligned to 2-byte boundary, and `int`, `long`, `float`, `double` must be aligned to 4-byte.
- This may cause data gaps in structures.
- Consider gaps to reduce RAM usage.

```
struct{
    char a;
    int b;
} info;
```



`scanf` Function

- `scanf` is a `<stdio.h>` function which does formatted input. Its first parameter is the format, and the second parameter is the address of data to receive input.
- `&` is the address operator, which takes the address of a variable.
- `%d` scans integer literal and put the result in int, `%lf` scans floating point literal and put the result in `double`.

Ex. 4.3 Unions (1)

```
#include <stdio.h>
struct{
    int tag;
    union{
        int i;
        double d;
    } val;
} data;

/* to be continued */
```

Ex. 4.3 Unions (2)

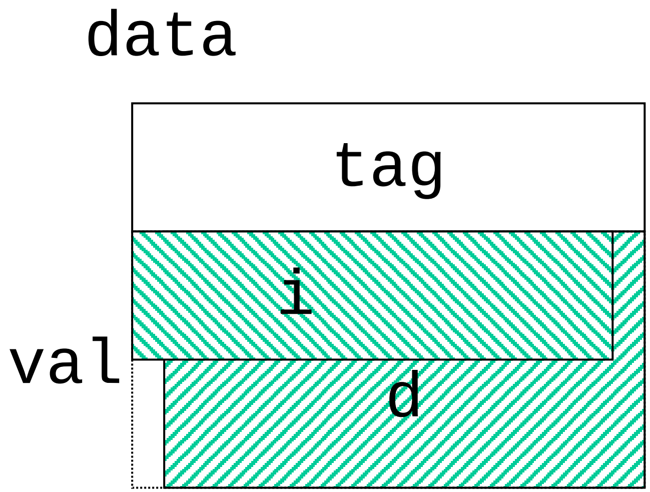
```
main()    /* continued */
{
    char c;
    scanf("%c", &c);
    while (c=='i' || c=='d'){
        if (c=='i'){
            data.tag=0;
            scanf("%d", &data.val.i);
        }
        else{
            data.tag=1;
            scanf("%lf", &data.val.d);
        }
        if (data.tag==0)
            printf("int value: %d\n", data.val.i);
        else
            printf("double value: %f\n", data.val.d);
        scanf(" %c", &c);
    }
}
```

Unions

- Unions is like a structure, but the members are allocated at the same address and shares memory area.

```
struct{
    int tag;
    union{
        int i;
        double d;
    } val;
} data;
```

Member declarations



Notes on Union

- The union data should be references with the same member name as it is assigned. <u>Don't access union members through a different member name.</u>
- It is recommended to have some member that remembers which member of the union has been stored (`tag` in this case).

More on `scanf`

- `%c` inputs character and assigns it to `char` data.
- Space character in format string indicates scans over the invisible (space and newline) characters.

Ex. 4.4: Pointers (1)

```
#include <stdio.h>
#include <stlib.h>
struct node{
  int val;
  struct node *next;
} *root;

/* to be continued */
```

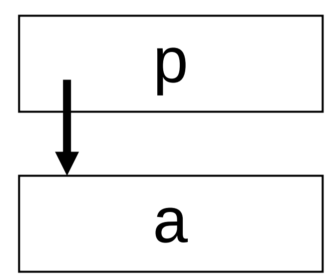
Ex. 4.4: Pointers (2)

```
/* continued */
main()
{
    int i;
    struct node *p;
    root=NULL;
    scanf("%d\n", &i);
    while (i>=0){
        p=malloc(sizeof(struct node));
        p->val=i;
        p->next=root;
        root=p;
        scanf("%d", &i);
    }
    p=root;
    while (p!=NULL){
        printf("%d\n", p->val);
        p=p->next;
    }
}
```

Pointers

- Pointer data holds memory address, but it must be declared to point to some specific data type.
- You can take address of any data in memory by & operator.
- You get the contents of the pointer by `*` operator.

`int *p;`	Declares that `p` points `int`.
`int a;`	
`p=&a;`	`p` points `a`.
`a=1;`	
`printf("%d\n", *p);`	Prints the value of `a`.



Notes on Pointers

- Pointers must be initialized with some address, otherwise, it points to indefinite location. <u>Don't use uninitialized pointers.</u>
- `NULL` is the pointer constant with value 0 (defined in `<stdlib.h>`: basic library). We have a convention that "`NULL` points to nothing".
- <u>Don't de-reference `NULL` pointers.</u>

Structure Names

- `struct node{ ... }` gives the name `node` to the structure.
- Once you give a name to a structure, you can declare the same structure without specifying members.
- Structure name is necessary when you declare self-referencing structure.

```
struct s{
  int x, y;
};
```

Declaration of a structure

```
struct s a, b;
```

Declaration of variables with the type `struct s`.

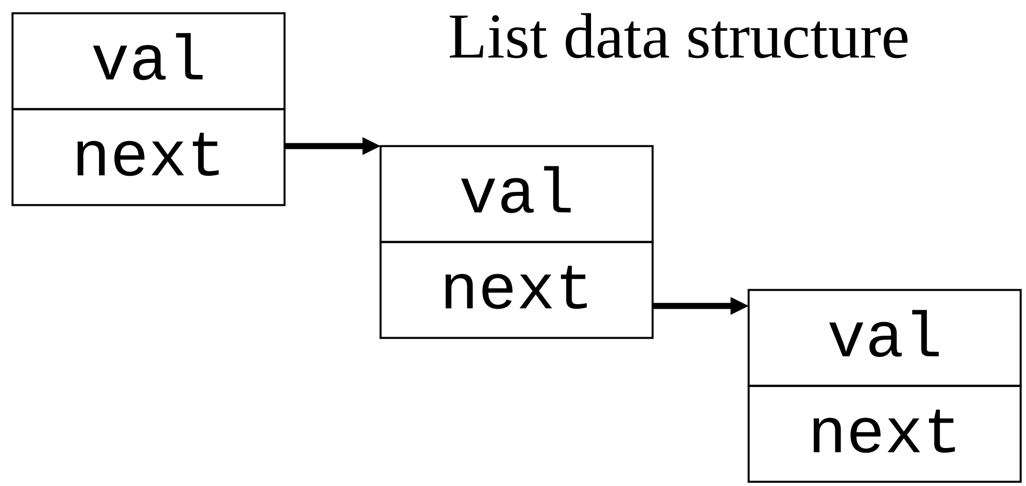
Operator `->`

- Operator `->` refers to the data which the pointer member points to.
- `a->b` can be considered as an abbreviation of `((*a).b)`.

Self Rererencing Structures

- Structure cannot include the same structure, but can include the pointer to the same structure. This is used to define useful data types.
- To include the reference to itself, the structure needs the name.

node

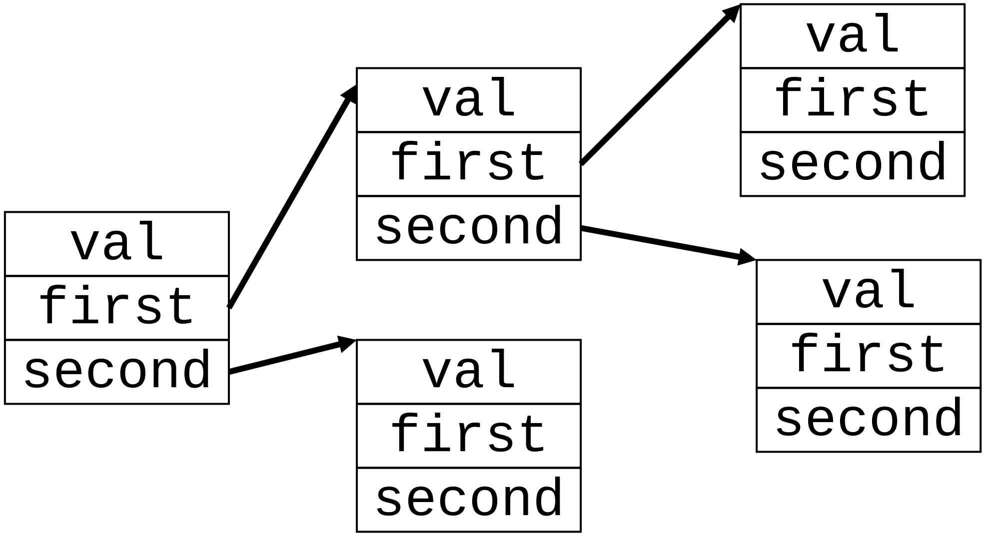


List data structure

```
struct node{
  int val;
  struct node *next;
} *root;
```

More Data Structure Example

```
struct tree_node{
  int val;
  struct tree_node *first, *second;
} *root;
```



`malloc` Function

- `malloc` function is memory allocation function defined in `<stdlib.h>`.
- Memory allocation library must be implemented on your system to use `malloc`.
- `malloc` receives byte size of the area and returns the pointer to the newly allocated area.

Notes on `malloc` funciton

- In embedded programming, prefer static allocation (by global data) to dynamic allocation (by `malloc`).
- You can release allocated area by `free`, and these area can be re-allocated, but the memory area may be fragmented.
- <u>Allocate big persistent data (data which live long) first, to avoid fragmentation.</u>

`sizeof` Operator

- `sizeof` operator returns the byte size of the operand (type or variable).
- Useful to retrieve the characteristics of the compiler: e.g.
  `if (sizeof(int)==4)` ...

Pointer Arithmetic

- When adding/subtracting integer to pointer, the integer is multiplied by the size of the pointed data. e.g. `long *p; p+1` adds 4 to the address value `p`.
- When pointer to the same type is subtracted, the difference is divided by the size of pointed data.
- Thus, `p[i]` is an abbreviation of `*(p+i)`.
- The same rule applies to `++` and `--`.

5. Function Interface

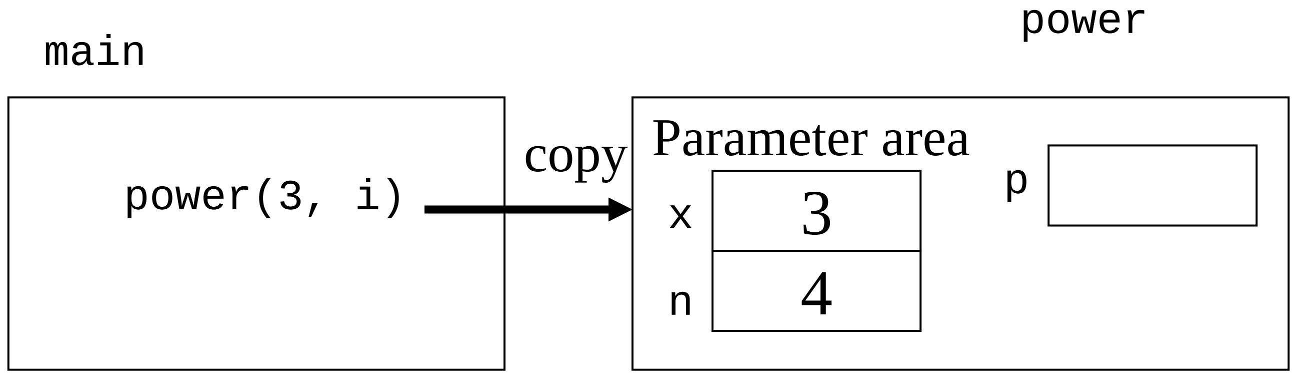
- Function parameters are copied and passed to the function.
- Modifying parameters doesn't affect the original parameters.

Ex. 5.1: Parameter Passing

```
#include <stdio.h>

int power(int x, int n)
{
    int p;
    p=1;
    while (n>0){
        p=p*x;
        n--;
    }
    return p;
}

main()
{
    int i;
    i=4;
    printf("%d\n", power(3, i));
}
```

Function Execution Model

- Function parameters are copied (to registers or stack) and passed to the function.
- Function allocates its local data are allocated to the stack, and the area is deallocated when function exit.



Ex. 5.2: Pointer Parameters

```
#include <stdio.h>

void swap(int *p, int *q)
{
    int temp;
    temp=*p;
    *p=*q;
    *q=temp;
}

main()
{
    int i, j;
    i=1;
    j=2;
    swap(&i, &j);
    printf("%d %d\n", i, j);
}
```

Pointer Parameters

- Use pointer parameters in the following cases:
  - Data to be passed is too big and copying is not efficient.
  - You want to change the data in the calling function by the called function.
  - You want to pass a function as a parameter (next example).
- As the parameters are copied, you cannot change the original value by changing a parameter in a called function.

Ex. 5.3: Passing Functions

```
#include <stdio.h>

int square(int n)
{
    return n*n;
}

int sum(int (*p)(int))
{
    int i, s;
    s=0;
    for (i=0; i<10; i++)
        s=s+(*p)(i);
    return s;
}

main()
{
    printf("%d\n", sum(&square));
}
```

Function Parameters

- Pointer to functions can be declared as data, or passed as parameters.
- The function pointed can be called using `*` operator (e.g. `(*p)(i)`).

Function Prototypes

- Function prototype is a declaration of a function (without definition) specifying parameter types.
  e.g. `int f(int, int *);`
- Use function prototypes to check function caller-callee interface.
- Don't call a function before function prototype declaration.

6. Topics on Data Types

6. Topics on Data Types

- Complex type declarations
- Typedefs
- `const` and `volatile`
- `enum` types

How to Declare Pointer to a Function

- `int (*p)(int)` declares a pointer to a function (which receives `int` parameter).
- This can be interpreted as follows:
  - `int X(int)` declares `X` to be a function receiving `int` parameter.
  - Replacing `X` by `(*p)` (note the precedence) declares `*p` to be a function receiving `int` parameter.
  - This shows that `p` is a pointer to a function receiving `int` parameter.

Other Complex Declarations

- `int *p(int)` (interpreted as `int *(p(int))`) declares `p` as a function (receiving `int` parameter) which returns a pointer to `int`.
- `int (*p)[5]` declares `p` as a pointer to an array of 5 `int`'s.
- `int *p[5]` declares `p` as an array of 5 pointers to `int`.
- All these interpretations can be verified by considerint how `p` is used in an expression.

Declaration of Types

- Types are recursively defined as follows:
  - Basic types (int, long, short, char, float, double, struct, union, etc.) are types.
  - Let T be a type, then "pointer to T" is a type.
  - Let T be a type, then "n element array of T" is a type.
  - Let T be a type, then "function returning T" is a type (actually, parameters must be considered, but here we use simple version).
- Example:
  10 element array of (pointer to (function returning `int`))

Declaration and type

- Type declaration can be recursively derived as follows:
  - If T is declared as ... T ..., then PT: "pointer to T" is declared as ... (*PT) ...
  - If T is declared as ... T ..., then AT: "n element array of T" is declared as ... (AT[n]) ...
  - If T is declared as ... T..., then FT: "functino returning T" is declared as ... (FT()) ...
- Example:
  - FI: function returning int: `int FI();`
  - PFI: pointer to (function returning int): `int (*PFI)();`
  - APFI: 10 element array of (pointer to (function returning int)): `int (*(APFI[10]))();`

Declaration of types: Example

- Another way to find declaration of complex type is consider what expression gives you the basic type from the type to be declared.
- To declare a variable `p` to be of type
  10 element array of (pointer to (function returning `int`))
  - `p` is 10 element array of (pointer to (function returning `int`))
  - `p[i]` is pointer to (function returning `int`)
  - `*p[i]` (`*(p[i])` when fully paranthesized) is function returning `int`
  - `*p[i]()` is `int`
  - `p` is declared as `int *p[10]();`
- Apply type constructor to the declared name in the reverse order of its occurrence in the description.

Construction of Type Declaration with `typedef`

- You can do type construction step-by-step using `typedef`

```
typedef int FI(void);     /* FI is function returning int */
typedef FI *PFI;          /* PFI is pointer to FI       */
typedef PFI APFI[10];     /* APFI is 10 element array of  */
                          /* PFI                          */
APFI p;
```

`const` and `volatile`

- `const` data cannot be assigned.
  - Usually used for ROM data, but also can be used for parameters/local variables/structure members, so that compiler can check if they are modified.
- `volatile` data are guaranteed to be loaded and stored from/to memory whenever they appear in the program (i.e. compiler doesn't optimize load/store).
  - Used for I/O registers, and data which might be modified by interrupt processing.
- A data can be `const` and `volatile` at the same time (e.g. timer register)

`const` (`volatile`) pointers

- `const int i=10;` declares `i` to be read-only `int` data.
- `const int *pci;` declares `pci` to be a pointer to `const int`.
  - `pci` itself can be modified
  - `*pci` (the data which `pci` points to) cannot be modified.
- To declare non-modifiable pointer-to-`int` data, use `int` `* const` `cpi=&i;`
  - "`* const`" is a type constructor to build constant pointers.
- `volatile` and `const volatile` has similar syntax.

`enum` types

- Enumeration declares enumerated data type.
  - `enum DAY {sun, mon, tue, wed, thu, fri, sat};`
- Enumeration members can be used whenever `int` can be used.
- Each member are assigned consecutive value starting from 0 (`sun`=0, `mon`=1, ..., `sat`=6, etc.).
- Use `enum` type for enumerated items for which assignment of value is arbitrary (like `enum DAY` above).
- Usage of `enum` is more readable than using small numbers (0, 1, 2, ...) directly.

Another use of `enum`

- `enum` members can be assigned explicit values.
  - `enum LIMITS{max_buf=128, max_name=32};`
- Better way to define integer constants than `#define`.
- Debugger usually doesn't recognize `#define` constants (because they are expended before compilation), but can recognize `enum` members.

7. Compiler Directives

- Compiler directives gives a command to the compiler.
- `#define` defines a macro.
- `#ifdef` selects the compiled portion of the program.
- `#include` allows file inclusion.
- The grammar of directives are independent of C language syntax, and C syntax applies after the directives are processed.
- Compiler directives are processed line by line. The line starting with # introduces a compiler directive.

Ex. 7.1: Macro Definitons

```
#include <stdio.h>

#define PI 3.14

#define AREA(r) (PI*(r)*(r))

main()
{
    printf("%f\n", AREA(2.0));
}
```

Macro Definitions

- `#define` defines a macro with or without parameters.
- `#define` name text defines a parameterless macro, which indicates that text replaces name.
- `#define` name(parameters) text defines a macro with parameters, and parameter names in the text is replaced by the actual parameters in the macro call.

Macro Replacement

```
AREA(2.0)
```

Macro `AREA` is applied

```
(PI*(2.0)*(2.0))
```

Macro `PI` is applied

```
(3.14*(2.0)*(2.0))
```

Notes on Macros

- When declaring macros with parameters, `(` (the opening parenthesis starting macro parameters) must be written immediately after the macro name.
- <u>Enclose macro parameters and replacement text by parentheses to avoid unexpected interpretation</u>

```
#define X 1+2
...
X*3
...
```

Expanded as `1+2*3` which is interpreted as `1+(2*3)`

Notes on Macros

- Don't specify an expression with side effects in a macro call.
- The macro definition might use the parameter more than once, in which case, the side effect might be executed more than once.
- The number of side effects might vary when the macro definition is modified.

```
#define MAX(x, y) ((x)>(y)?(x):(y))
...
return MAX(a++, 1);
...
```

Expanded into
`((x++)>(1)?(x++):(1))`
`x++` might be executed more than once.

Ex. 7.2: Conditional Compilation

```
#include <stdio.h>

#define DEBUG

sub(int i)
{
#ifdef DEBUG
    if (i<0)
        printf("Bad Argument: %d\n", i);
#endif
    printf("%d\n", i*i);
}
```

Conditional Compilation

- Conditional compilation can select the part of the program to be compiled.
- The part enclosed by `#ifdef` name ... `#endif` is compiled only if the name is defined.
- Useful to keep track with software versions, or insert debugging statements.

Ex. 7.3: File Inclusion

file1: def.h

```
extern int a;
extern void sub(void);
```

file2: main.c

```
#include "def.h"
int a;

main()
{
    sub();
}
```

file3: sub.c

```
#include <stdio.h>
#include "def.h"

void sub(void)
{
    a=1;
    printf("%d\n", a);
}
```

File Inclusion

- `#include "filename"` includes user-defined file (searches from the same directory as the compiled file).
- `#include <filename>` includes system-defined file (searches from the system directory).

Dividing a Project into Files

- Keyword `extern` specifies a declaration without defining an object/function.
- Put `extern` declaration in a common header (`.h`) file.
- Each C (`.c`) file include the common header file.
- Make sure that each object/function is defined in exactly one of the C files.

`static` Declarations

- When declared with `static` keyword (instead of `extern`), the function or variable is local to the file, and cannot be accessed from other files.
- <u>Use `static` to define local variables and funcions.</u>

II. Embedded Programming in C

Contents

1. Introduction

- This part explains how a C program works in the embedded environment.
  - Introduction to Assembler
  - Correspondence begween C program and ROM/RAM
  - Program Startup and Interrupt Handling
  - Accessing Hardware
  - Linkage with assembler program

Difference of Embedded Programs from PC/Workstation Programs

- Explicit memory configuration (ROM, RAM, I/O registers)
- Initialization (from RESET to main)
- No stdio (unless you write it yourself)
- Processing is driven by interrupts
- Program should run permanently (higher quality, error recovery required)

2. Introductio to SH Assembler

- In embedded programs, knowledge of assembler is helpful for
  - Writing operations not supported by C
  - Understanding hardware behavior
  - Tuning up programs (extracting full performance of the CPU)
  - Debugging optimized C code

Use of Assembler Programs "To C or not to C"

- The following operations cannot be written in standard C:
  - Reading/Writing special registers (e.g. Stack Pointer)
  - Interrupt handling (a process returning with RTE instruction)
- Some compiler have language extensions to do this (check SH C Compiler Manual). But they are not portable.
- We recommend you to first understand what is happening with assembler programs, and then use non-portable features of each compiler.

Assembler Syntax

- Instructions

[<label>:] <operation> [<operand>[,<operand>]...]

e.g.

```
func:   MOV.L    R4,R0
        ADD      #2,R0
```

- Labels starts at the beginning of line, and represents the location (address) of the instruction
- First operand is source, second operand is destination

Assembler Directives

- Directives (operation starting with ".")
  - `.DATA` allocates data

    ```
    .DATA.L H'100
    .DATA.B H'FF
    ```

    `.L`, `.W`, `.B` specifies the size of data (4, 2, 1 byte, respectively)
  - `.RES` reserves area

    ```
    .RES.L H'100
    .RES.B H'FF
    ```

    Specifies memory area to be reserved. Unit is 4, 2, 1 byte, according to its size.
  - `.SECTION` defines a section (contiguous memory area) and its attribute

    ```
    .SECTION P,CODE,ALIGN=4
    .SECTION C,DATA,ALIGN=4
    ```

    `P`, `C` are the name of the sections. Other operands specifies the attribute of the section.
  - `.END` specifies the end of the program.

Assembler Directives

- `.IMPORT` refers to a label defined in another module.

```
.EXPORT   _a
```

- `.EXPORT` makes the label available from other modules.

```
.IMPORT   _a
```

- `.ALIGN` alignes next instruction/data.

```
.ALIGN    4       ; aligns next data to 4-byte
                  ; boundary
```

Assembler Program Structure

```
     .EXPORT  _f                   ; External declaration of labels
     .SECTION P,CODE,ALIGN=4       ; Start of program
_f:  MOV.L    R4,R0                ; _f is the function entry
     MOV.L    L,R1
     ADD      R1,R0
     RTS                           ; delayed branch
     NOP
     .ALIGN   4                    ; Alignes next data to 4-byte
L:   .DATA.L  LABEL
     .SECTION C,DATA,ALIGN=4       ; Start of data
LABEL:
     .DATA.L  H'1000
     .END                          ; End of program
```

Programming in SH Assembler

- Load/Store Architecture
  - SH is a RISC. Only `MOV` instructions can access memory. Other operations (e.g. `ADD`) are performed between registers, or small constant and register.

```
MOV.L  @R4,R0
MOV.L  @R5,R1
ADD    R0,R1
ADD    #1,R1
MOV.L  R1,@R4
```

Programming in SH Assembler

- Addressing Modes
  - Addressing modes specifies the location of an operand. The following addressing modes are frequently used.

`#immediate`	Constant value.
`Rn`	Register
`@Rn`	Register indirect (`*` operator in C)
`@Rn+`	Like `@Rn`, but increment `Rn` (by operand size) after the access, used to pop data from the stack
`@-Rn`	Like `@Rn`, but increment `Rn` before the access, used to push a data into a stack
`@(disp,Rn)`	"disp" bytes from the address specified by Rn
`@(R0,Rn)`	Add R0 and Rn to get the operaand address (array indexing)
`@(disp,PC)`	PC relative: "disp" bytes from PC (the address of current instruction). Can also be specified by a label in the program

Programming in SH Assembler

Registers

Type	Registers	Initial Value*
General registers	R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, R8–R15	Undefined
Control registers	SR	MD bit = 1, RB bit = I3–I0 = 1111 (H'F), r undefined
	GBR, SSR, SPC, SGR, DBR	Undefined
	VBR	H'00000000
System registers	MACH, MACL, PR, FPUL	Undefined
	PC	H'A0000000
	FPSCR	H'00040001
Floating-point registers	FR0–FR15, XF0–XF15	Undefined

Note: * Initialized by a power-on reset and manual reset.

General Registers (R0-R15) can be used in MOV or other operations (R15 is a stack pointer)

Control Registers are transferred using LDC/STC instructions.
SR (status register) holds flags.

System Registers are transferred using LDS/STS instructions.
PC is a program counter.
PR holds the return address of a function.
MACH, MACL holds result of multiplication.

Programmin in SH Assembler

- Literal Pool: How to load large constants?
  - SH instructions have 16-bit fixed format. So they cannot include 16/32-bit constants.
  - These constants should be located in a program (after unconditional jump instructon so as not to interfere program execution), and should be loaded using PC-relative addressign mode.

Programmin in SH Assembler

Literal pool example:

```
    MOV.L    f_addr,R0    ; Load constant _f
    JSR      @R0          ; Calls function _f
    NOP
    MOV.W    dat1,R1      ; Load constant H'100
    ADD      R1,R0
    RTS
    NOP
; Start of Literal Pool (after unconditional branch)
    .ALIGN   4            ; Don't forget to align data
f_addr:
    .DATA.L  _f           ; Address of function "_f"
    .ALIGN   2
dat1:
    .DATA.W  H'100
```

Programming in SH Assembler

- Delayed Branch
  - When branch instruction is executed, CPU must wait until the instruction from branch target is fetched.
  - Delayed branch mechanism executes the next instruction, which is already fetched when the branch instruction is executed.
  - The instruction next to the branch instruction is called the instruction in "delay slot".
  - There are restrictions for instructions in "delay slot", for example, you cannot put an instruction using PC in delay slot.
  - If you cannot put an instruction in delay slot, you should put `NOP` in the delay slot.
  - Typical delayed branch instructions are: `BRA, BSR, JMP, JSR, RTS, RTE.`

Programmin in SH Assembler

Delayed branch example:

```
_f:     BSR     _g
        MOV     #1,R4   ; Delay slot for BSR
                        ; Passes parameter 1 through R4
        RTS
        NOP             ; Delay slot for RTS
_g:     MOV     R4,R0   ; R0 holds return value
        RTS
        ADD     #1,R0   ; Delay slot for RTS
                        ; Increments the return value
```

Programming in SH Assembler

- Comparison
  - SH have only one flag (T) to indicate comparison result. We have several comparison instructions for various comparison operation.
  - BT or BF instruction is used to jump according to comparison result.

CMP/EQ	#imm,R0	When R0 = imm, 1 → T Otherwise, 0 → T	10001000iiiiiiii	—	Comparison result
CMP/EQ	Rm,Rn	When Rn = Rm, 1 → T Otherwise, 0 → T	0011nnnnmmmm0000	—	Comparison result
CMP/HS	Rm,Rn	When Rn ≥ Rm (unsigned), 1 → T Otherwise, 0 → T	0011nnnnmmmm0010	—	Comparison result
CMP/GE	Rm,Rn	When Rn ≥ Rm (signed), 1 → T Otherwise, 0 → T	0011nnnnmmmm0011	—	Comparison result
CMP/HI	Rm,Rn	When Rn > Rm (unsigned), 1 → T Otherwise, 0 → T	0011nnnnmmmm0110	—	Comparison result
CMP/GT	Rm,Rn	When Rn > Rm (signed), 1 → T Otherwise, 0 → T	0011nnnnmmmm0111	—	Comparison result
CMP/PZ	Rn	When Rn ≥ 0, 1 → T Otherwise, 0 → T	0100nnnn00010001	—	Comparison result
CMP/PL	Rn	When Rn > 0, 1 → T Otherwise, 0 → T	0100nnnn00010101	—	Comparison result

Programming in SH Assembler

Comparison example (if statement)

```
        CMP/GT R4,R5   ; Compare R4 and R5
        BT     L1      ; if R5>R4, go to L1
        MOV    R4,R0   ; R0=R4
        BRA    L2      ; go to L2
        NOP
L1:     MOV    R5,R0   ; R0=R5
L2:
```

Of course, you can eliminate NOP after BRA, by moving previous MOV instruction to delay slot.

Programming in SH Assembler

Comparison example (for loop)

```
        MOV    #10,R1   ; R1=10
L1:     CMP/EQ #0,R1    ; Compare R1 with 0
        BT     L2       ; if R1==0 then exit loop
        JSR    _f       ; Call subroutine f
        NOP
        SUB    #1,R1    ; R1=R1-1
        BRA    L1       ; Repeat
        NOP
L2:
```

Programming in SH Assembler

- How to write a function in Assembler:
  - Parameter convention (defined by C)
    - Parameters: R4-R7 (up to 4 parameters on registers)
    - Return value: R0
  - Register saving/restoring
    - R8-R14 must be saved and restored if used in the function
    - PR holds the return address (set by JSR instruction). PR must be saved and restored when the function calls another funtion.

Programming in SH Assembler

Function example (Empty function)

```
;; Empty Function (does nothing)
_f:                 ; "_" is added before C identifier
                    ; In C, call f()
          RTS       ; Jump to the address indicated by
                    ; PR, which is the PC when this
                    ; function is called.
          NOP       ; Delay slot
```

Programming in SH Assembler

Function example (Parameters and Return Value)

```
;; Returns the sum of two parameters
_sum:                  ; From C, parameters are passed
                       ; via R4 (parameter 1),
                       ;     R5 (parameter 2).
        ADD  R5,R4 ; Noew R4 holds the sum
        MOV  R4,R0 ; Return value is set in R0
        RTS
        NOP
```

Programming in SH Assembler

Function example (Register saving/restoring)

```
_g:     MOV  R8,@-R15   ; Save register R8 and R9
        MOV  R9,@-R15   ; on Stack.
                        ; Register R8-R14 must be
                        ; saved if it is used, by C
                        ; calling convention
        STS  PR,@-R15   ; Save PR if the function calls
                        ; another function.
        ...
        JSR  _f         ; Calls another function
        NOP
        ...
        LDS  @R15+,PR   ; Restores registers
        MOV  @R15+,R9   ; Note that regsiters are
        MOV  @R15+,R8   ; restored in reverse order
        RTS
        NOP
```

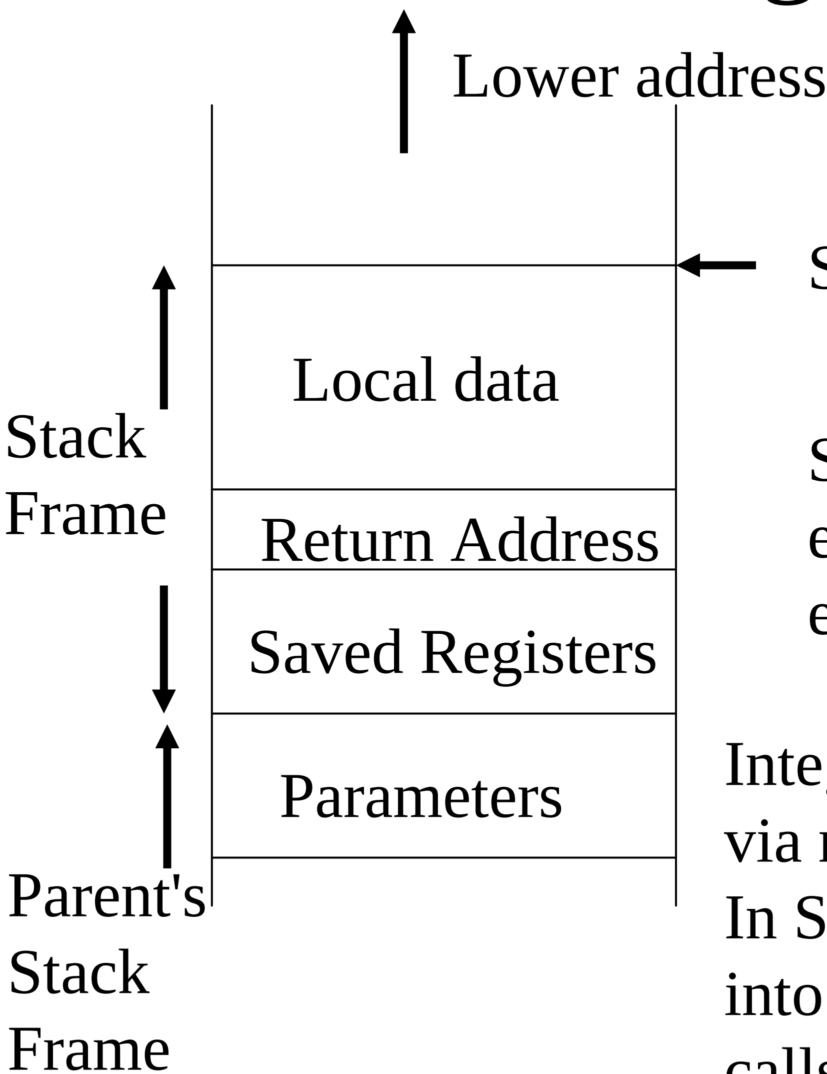
Assembler Program Comments

- Assembler programs are more difficult to read. So you should write more comments in the program.
  - Comment on every line.
  - Comment on register usage.
  - Comment on function interface (interface registers).
  - Comment on flag usage (if used).

3. C Program Memory Model

- C program uses following kinds of memory areas:
  - Program code (initialized, read only)
  - Constant data (initialized, read only)
  - Initialized data (initialized, read/write)
  - Uninitialized data (uninitialized, read/write)
  - Stack (used for function-call interface, parameters, and local data)
  - Heap (managed by library functions: malloc, etc.)

Usage of Stack



Stack Frame is built on function entry, and deallocated on function exit.

Integer parameters (up to 4) are passed via registers in SH.
In SH, the return address may not be pushed into the stack, if there is no further function calls

4. Sections

- Sections are relocatable (i.e. can be placed anywhere in the memory) unit of program or data.
- Each C program compilation unit generates 4 kinds of sections.
- The same kind of sections from several compilation unit are linked together in a contiguous memory area.

Section Attributes

Code	Section name	Attribute	Initialization	Memory
`int a;`	BSS	R/W	Zero	RAM
`int b=1;`	Data	R/W	Initialized	RAM
`const int c=1;`	Const	R	Initialized	ROM
`main(){ ... }`	Text	R	Initialized	ROM

```
int a;
int b=1;
const int c=1;
main(){
    ...
}
```

BSS is an abbreviation of "Block Storage Segment"

`const` type

- `const` keyword specifies that the data cannot be assigned.
- The data with `const` type can be placed in ROM.
- `const int *p;` declares that `p` points to constant area, but `p` can be assigned. To declare const pointer, use declaration

```
int *const p;
```

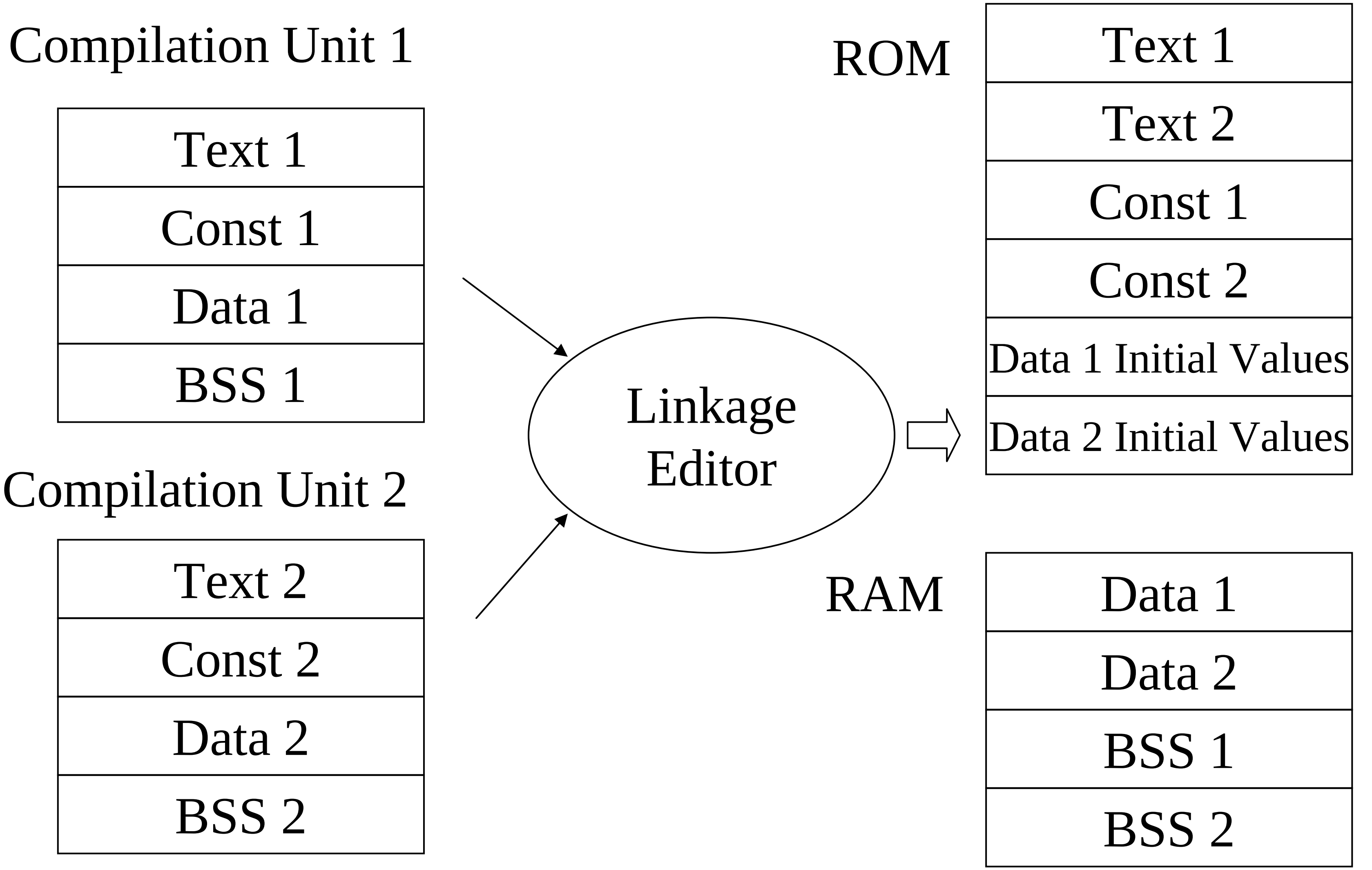
Data Section and its Initialization

- Data section has its initial value, but the variables in data section can be modified.
- To implement this, data section must be allocated in RAM, but its initial value must be copied from ROM at program startup.

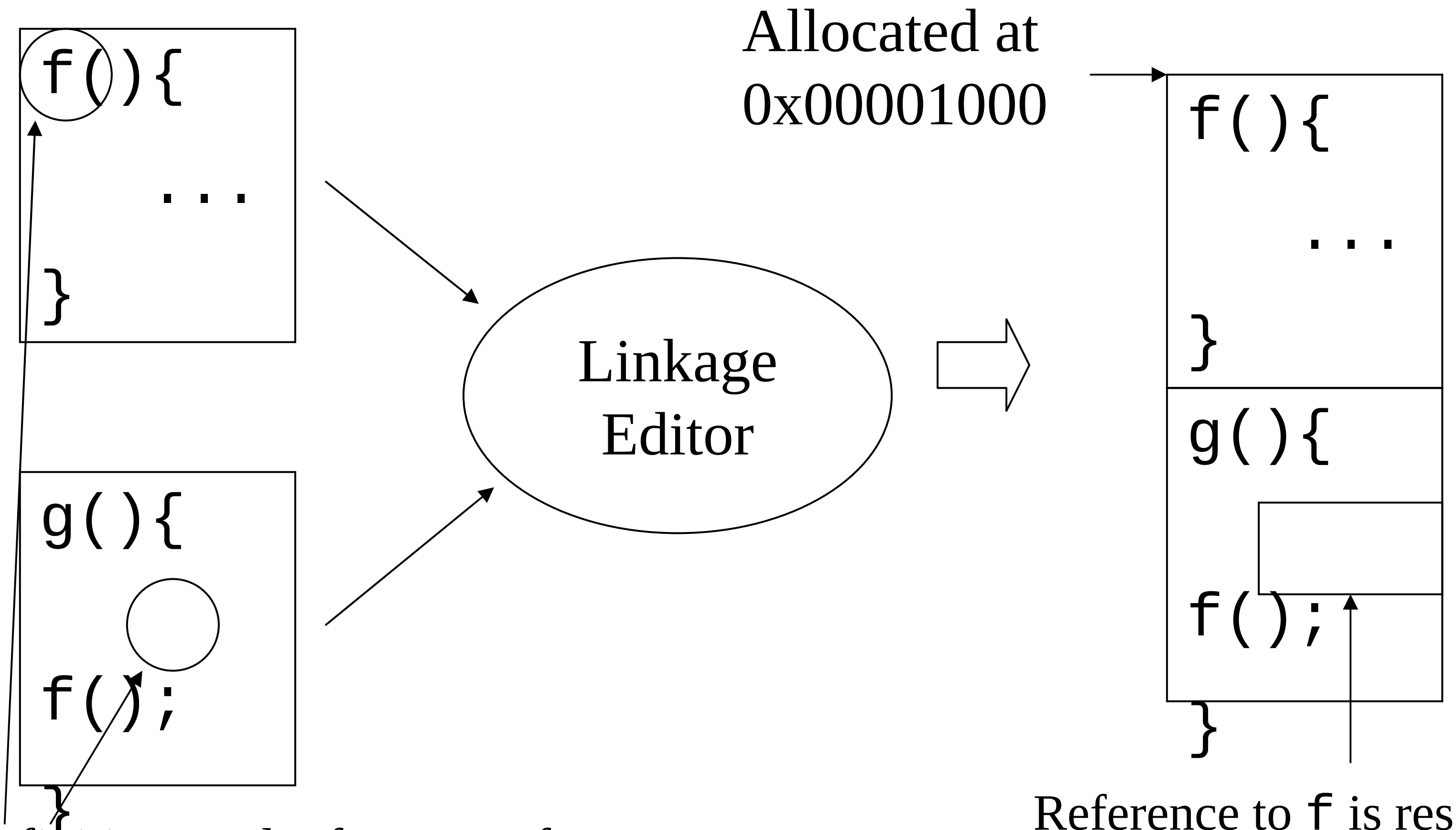
Functions of Linkage Editor

- The linkage editor collects the sections of the same name from several compilation units, and allocates them in a contiguous area.
- The linkage editor resolves the references to variables/functions by the allocated addresses.

Allocation of Sections



Resolving References



Definitions and references of functions/variables are retained symbolically.

Reference to `f` is resolved by the address 0x00001000

Assignment of Absolute Address to Sections

- To start up a program, you need to assign fixed absolute address to the entry point of the program.
- Linkage editors can assign absolute address to a section (e.g. `START sec1(100)` allocates the section `sec1` to the address 100 in Renesas Linkage Editor).
- Locate the entry point at the beginning of a compilation unit.
- Link the module including the entry point as the first member of a section.
- Use linkage editor command to allocate the section to the specific address.

5. Program Startup and Interrupt Handling

- The following assumes SH4 architecture.
- Program starts at the address 0xa0000000.
  - Check exception event, and jump to appropriate routine.
  - The exception event is power-on-reset, go to overall initialization.
- When interrupt, the program goes to the address VBR+0x600.
  - Check interrupt event and call appropriate interrupt handler.

Reset Handler

```
_ResetHandler:                          ; Located at 0x80000000
        mov.l   #EXPEVT,r0
        mov.l   @r0,r0
        shlr2   r0
        shlr    r0
        mov.l   #_RESET_Vectors,r1
        add     r1,r0
        mov.l   @r0,r0
         mov.l  #_INIT_SP,r15           ; Initialize Stack Pointer
        jmp     @r0
        nop
```

Tables for Exeption/Interrupt Handlers

These labels can be C function names

```
_RESET_Vectors:
        ;H'000 Power On Reset
        .data.l  _PowerON_Reset
        ...                          ; Other reset handler addresses

_INT_Vectors:
        ...                          ; Other interrupt handler addresses
        ;H'1C0 NMI                   ; Here the offset should be 0x1C*2
        .data.l  _INT_NMI
        ;H'1E0 User Break
        .data.l  _INT_User_Break
        ;H'200 External hardware interrupt
        .data.l  _INT_Extern_0000
        ...                          ; Other interrupt handler addresses
```

Interrupt Handler (1)

```
IRQHandler:                                       ; located at VBR+0x600
        PUSH_EXP_BASE_REG                         ; macro saving all the registers
        mov.l    #INTEVT,r0                       ; set event address
        mov.l    @r0,r1                           ; set exception code
        mov.l    #_INT_Vectors,r0                 ; set vector table address
        add      #-(h'40),r1                      ; exception code - h'40
        shlr2    r1
        shlr     r1
        mov.l    @(r0,r1),r3                      ; set interrupt function addr
        mov.l    #_INT_MASK,r0                    ; interrupt mask table addr
        shlr2    r1
        mov.b    @(r0,r1),r1                      ; interrupt mask
        extu.b   r1,r1
        stc      sr,r0                            ; save sr
        mov.l    #(RBBLclr&IMASKclr),r2           ; RB,BL,mask clear data
```

Interrupt Handler (2)

```
        and     r2,r0                   ; clear mask data
        or      r1,r0                   ; set interrupt mask
        ldc     r0,ssr                  ; set current status
        ldc.l   r3,spc
        mov.l   #__int_term,r0          ; set interrupt terminate
        lds     r0,pr
        rte
        nop
;       Interrupt terminate
__int_term:
        mov.l   #MDRBBLset,r0           ; set MD,BL,RB
        ldc.l   r0,sr                   ;
        POP_EXP_BASE_REG
        rte                             ; return
        nop
```

Initialization Program

```
void PowerON_Reset(void)
{
       /* Set Vector Base Register */
       set_vbr((void *)((_UINT)INTHandlerPRG - INT_OFFSET));
       _INITSCT();
       _INIT_IOLIB();        /* Initialize library */
       HardwareSetup();      /* Setup Hardware     */
       set_cr(SR_Init);       /* Set CR (be user mode) */
        nop();
       main();               /* Initialize applications */
       sleep();              /* Sleep to wait interrupt */
}
```

Definition of Sections

```
         .SECTION B,DATA,ALIGN=4
         .SECTION R,DATA,ALIGN=4
         .SECTION D,DATA,ALIGN=4
         .SECTION C,DATA ALIGN=4
;        D section contains initialized data (ROM) and R section
;        is its corresponding RAM area.
__B_BGN: .DATA.L (STARTOF B)
__B_END: .DATA.L (STARTOF B)+(SIZEOF B)
__D_BGN: .DATA.L (STARTOF R)
__D_END: .DATA.L (STARTOF R)+(SIZEOF R)
__D_ROM: .DATA.L (STARTOF D)
         .EXPORT __B_BGN
         .EXPORT __B_END
         .EXPORT __D_BGN
         .EXPORT __D_END
         .EXPORT __D_ROM
         .END
```

Section Initialization

```
extern int *_B_BGN, *_B_END, *_D_BGN, *D_END, *D_ROM;
void _INITSCT(void)
{
        register int *p, *q;
        /* Initialize BSS section to 0 */
        for (p=_B_BGN; p<_B_END; p++)
                *p=0;
        /* Copy initial values of Data section to RAM */
        for (p=_D_BGN, q=_D_ROM; p<_D_END; p++, q++)
                *p=*q;
}
```

6. Accessing Hardware

- Memory-mapped registers (I/O ports) can be accessed via pointers using their absolute addresses.

I/O Port Definitions

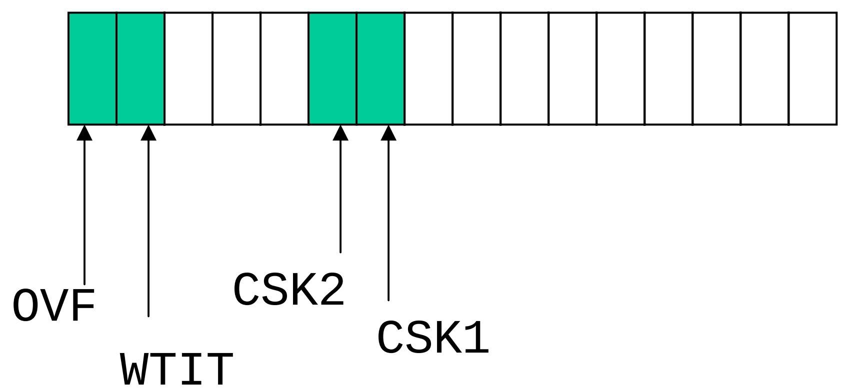
```
/* Timer Register                                         */
struct tcsr{
        unsigned short OVF:  1;   /* OVF  bit             */
        unsigned short WTIT: 1:   /* WTIT bit             */
        unsigned short :     3;   /* don’t care           */
        unsigned short CSK2: 1;   /* CSK2 bit             */
        unsigned short CSK1: 1;   /* CSK1 bit             */
        unsigned short :     9;   /* don’t care           */
};
#define TCSR_FRT (*(volatile struct tcsr *) 0x5FFFFB8)
...
```

Bit Fields

- Structure members declared with `:d` (d is the number of bits) are bit fields, and consecutive members are packed into its data size.
- Unnamed member indicates a gap in the bits.

```
struct tcsr{
        unsigned short OVF:  1;
        unsigned short WTIT: 1:
        unsigned short :     3;
        unsigned short CSK2: 1;
        unsigned short CSK1: 1;
        unsigned short :     9;
};
```

16 bit short data



`volatile` type

- The keyword `volatile` defines a data attribute (type) which requests that the compiler don't optimize the variable.
  e.g. `volatile int a;`

```
a=1;
a=2;
```

Compiler can eliminate the first assignment as an optimization. But such an optimization is not allowed for volatile type variables.

- `volatile` must be specified to declare I/O registers.

Cast (Type Conversion) Expressions

- The type name (`int`, `int *`, etc.) enclosed by `()` is called a cast operator, and used to convert the data to the indicated type.
- type name is a declaration without a name.
  e.g. `int *p` $\longrightarrow$ `int *`

Using Absolute Address to Access Hardware Registers

- `x=TSCR_FRT.OVF` reads `OVF` bit.
- `TSCR_FRT.OVF=1` sets `OVF` bit to 1.
- The expression `*(volatile struct tcsr *)0x5FFFFB8` can be interpreted as "convert absolute address 0x5FFFFB8 to a pointer to `volatile struct tcsr`, and access its contents".
- So the macro TSCR_FRT behaves just like a variable with type `volatile struct tcsr`.

7. Linkage with Assembler Programs

- Naming Conventions
  - Compiler attatches the character '`_`' in front of the data/function names of C.
  - To be linked, C names must be global, and assembler labels must be listed in `.export` directives. Assembler must list C names in `.import` directives.
- Calling Conventions
  - Save and restore registers R8-R14.
  - Parameters are passed through R4 (1st parameter) to R7 (4th parameter).
  - Return values are set in R0.

Example

```
extern int sub(int);

int a;

int f(void){
  return sub(1);
}
```

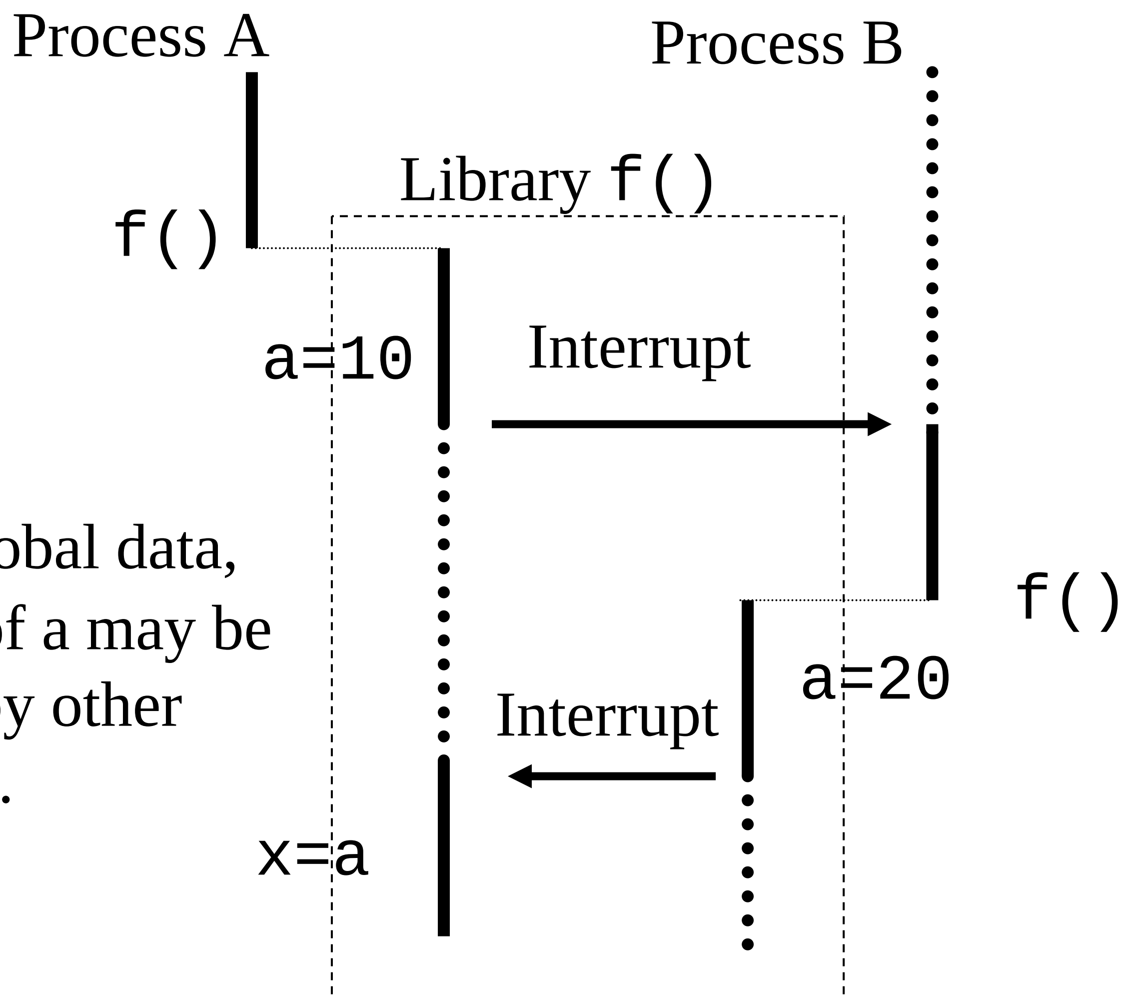
```
    .IMPORT     _a
    .EXPORT     _sub
    .SECTION    P,CODE,ALIGN=4
_sub:
    MOVE.L      #_a,R1
    MOVE.L      @R1,R0
    ADD         R4,R0
    RTS
    NOP
```

4/2 byte immediate are automatically converted into literal pool (to be generated after unconditional branch) by the assembler.

8. Re-entrant Library

- Library routines are shared among processes. They may be called from different processes at the same time.
- Library routines should use only local data (except references to `const` global data).
- Use synchronization primitives of OS (semaphores, etc.) to access global data or shared hardware resources.

Non-reentrant function



If `a` is a global data, the value of a may be modified by other proccesses.

III. Structured Program Design

Contents

1. Module Decomposition

When developing a large scale program, first decompose the program into functional unit.

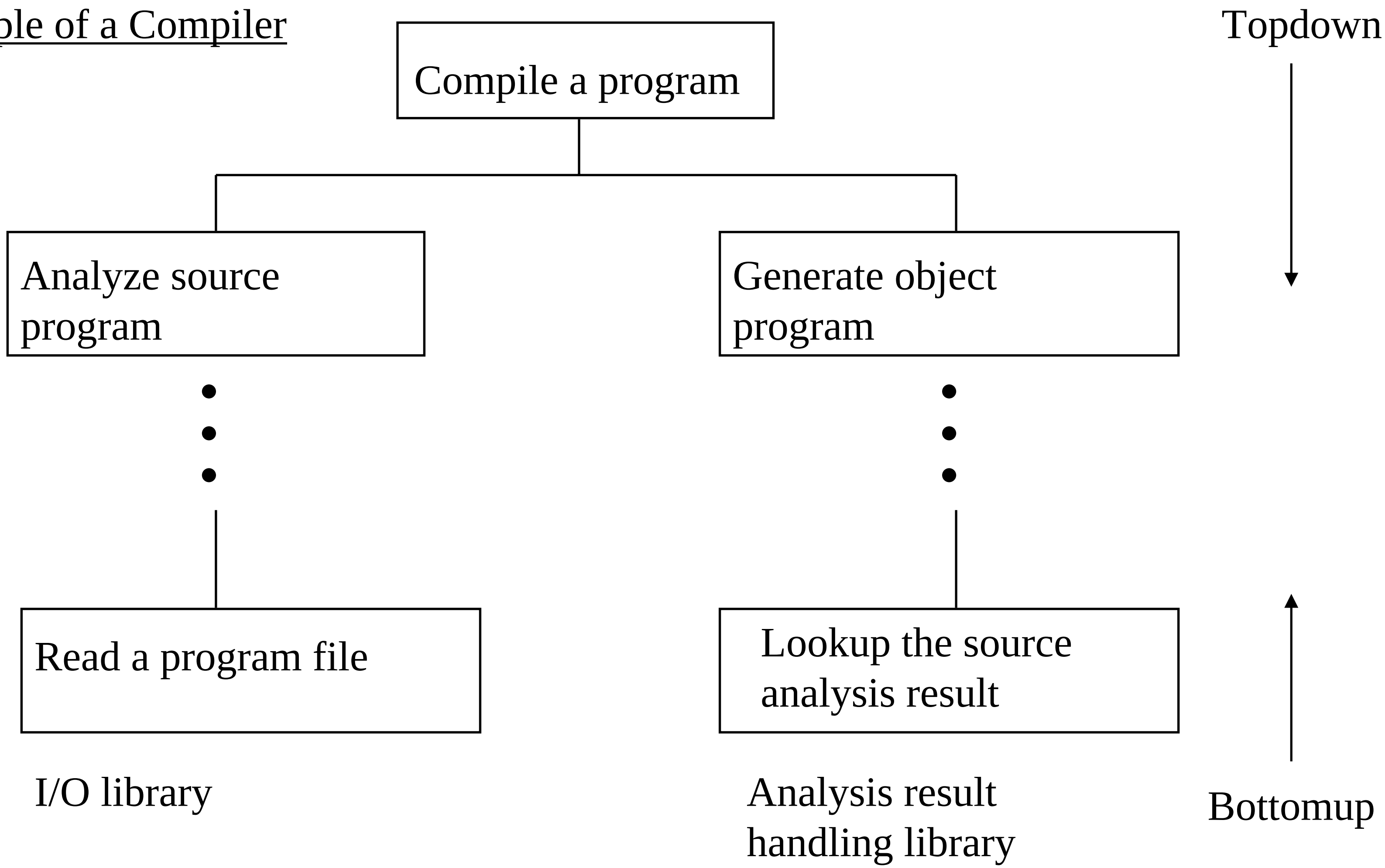
(1) A "Functional Unit" is a unit which you can state its function in one sentence. You should not decompose a program by execution sequence until you start detailed design.

(2) The "Topdown Design" decomposes the total program into smaller functional unit.

(3) The "Bottomup Design" designes basic set of functions, like I/O and handling of common data structures.

Sample Module Decomposition

Example of a Compiler



2. Topdown Design

Topdown analysis recursively decompose the program until you can start detailed design (design of concrete data structure and algorithm).

(1) For each decomposed module, write a header file, in which you write down the type declaration of the functions and data structure.

(2) The calling module include the header file of the called module, so that the interface definitions (functions and data structure) can be shared.

Topdown Design (Example)

main.c

```
#include "defs.h"
#include "anlyze.h"
#include "gen.h"
main(){
  anlyze();
  generate();
}
```

defs.h

Declares variables common to all the modules

anlyze.h

```
void anlyze(void);
```

generate.h

```
void generate(void);
```

anlyze.c

Definition of `analyze`

generate.c

Definition of `generate`

3. Bottomup Design

Bottomup design designes library modules, which accesses hardware, or handles common data structure.

(1) With only topdown design, similar algorithms appear in each modules.
Bottomup design avoids such a problem.

(2) Make the library modules more general, so that you can re-use the bottomup modules.

(3) Write a header file for each library module, and declare funcions and data structures in the library.

(4) The calling module of the library includes the header file of the library.

Sample Bottomup Design

anlyze.c

```
#include “io.h”
#include “tbl.h”
...
read_line();
...
enter_tbl();
```

Module calling library functions

I/O library

io.h

```
void read_line(void);
void write_line(void);
```

io.c

Definitions of `read_line`, `write_line`

Table handling library

tbl.h

```
void enter_tbl(void);
void remove_tbl(void);
```

tbl.c

Definitions of `enter_tbl`, `remove_tbl`

4. Implementing Modules in C

When programming in C, header ("`.h`") files (interface modules) and "`.c`" files (definition modules) are considered as modules.
Some modules don't have definition (e.g. modules with only #define directives).

<u>Structure of Modules</u>

(1) Header files (include only the declarations required to use the module)
  (a) `#define` directives required to use the module.
  (b) Type declarations (`struct`, `union`, `typedef`) required to use the module.
  (c) `extern` declarations of variables required to use the module.
  (d) Function declarations required to use the module.
(2) “.c" files
  (a) `#include` of the header files used in the module.
  (b) `#include` of the header of the module itself.
  (c) Local `#define`, `struct`, `union`, `typedef`s of the module.
  (d) `static` declarations of its local variables, and functions.
  (e) Definitions of its variables and functions.

`typedef` declarations

- Names declared with `typedef` keyword are type names.
- Use typedef declarations to declare frequently used types.

```
typedef int *POINTER;
```

Declare `POINTER` as the pointer to `int` type

```
POINTER p, *pp;
```

`p` is a pointer to `int`, and `pp` is a pointer to a pointer to `int`

Sample Module Implementation

stack.h

```
#define STK_SIZE 256

extern void init(void);
extern void push(int);
extern int pop(void);
```

stack.c

```
#include “stack.h”

static int sp;

static int stack[STK_SIZE];

void init(void){
  sp=0;
}

void push(int i){
  ...
}
...
```

Independence of Modules

A large scale program includes a lot of definitions and references to them.
To implement efficient development process, it is important to:

(1) Make definitions and references local to modules.
(2) Minimize references between different modules.

i.e. make modules as independent as possible.

Recommended Practice

(1) Decompose program so that a module implements a function which can be stated in one sentence.
(2) Declare functions and variables local to the module with `static` keyword.
(3) Make a header file for each module. Don't put everything in one header file.
(4) Don't include unnecessary header files.
(5) Design re-usable library modules.

Module Interface (Header Files)

Header files have two roles.

(1) A "specification" to define module interface.
(2) A mechanism with which a compiler checks the interfaces between modules.

<u>Recommended Practice</u>

(1) First write down the header file before writing the definition module.
(2) Use header files to review module interface.
(3) Don't change header file without reviewing with other team members.
(4) Refer to external functions/variables only through declarations in the header file.

Problems of Bypassing Header File Definitons

def.h

```
extern int x;
```

def.c

```
int x;
```

Reference using header file declaration (Recommended)

```
#include "def.h"

x=1;
```

Reference bypasssing header file declaration (bad)

```
extern int x;

x=1;
```

When the type of `x` has been changed, the reference will be illegal. The compiler cannot check the type incompatibility.

5. Good Programming Practice

- Programs should be easy to develop, easy to understand, easy to debug, and easy to maintain.
- Simple is the best.
- Documentation is very important.

Comments

- Comment at the beginning of every file (header comment), describing:
  - The project name
  - The name and the funciton of the module
  - Author
  - Revision history
- Comment on every data, describing its meaning.
- Comment on every function, describing:
  - The name of the function.
  - The function of the "function".
  - Meaning of its input (parameters) and return value.
  - The files and tables used or modified by the function.
- Comment on what you do, not on how (how you do it must be cleanly expressed by the program itself).

Example of a Header Comment

```
/*****************************************************************/
/* Copyright (c) 2005 by Renesas Technology Corp.,              */
/* All rights reserved.                                          */
/* Project name: SH C Compiler.                                  */
/* Module name:  gencode                                         */
/* Function:     Generate SH object code from intermediate*/
/*               language.                                       */
/* Author:       Yugo Kashiwagi                                  */
/* History:                                                      */
/* Aug. 01, 2005: Version 1.0                                    */
/* Aug. 30, 2005: Version 1.1                                    */
/*                Fixed register allocation bugs.                */
/* Sep. 22, 2005: Version 2.0                                    */
/*                Added new optimization.                        */
/*****************************************************************/
```

Example of Function Comment

```
/***********************************************************/
/* Function name:    search_table                          */
/* Function:         Search data in the table.             */
/* Input parameters: int key;        key data for search  */
/* Return value:     index of the table with the key      */
/* Table used:       key_table                             */
/* Table modified:   (None)                                */
/***********************************************************/
```

Naming Convention

- Use descriptive names for global variable/function names.
  e.g. `read_one_line`,
  `symbol_table_index`
- You can use simple names for local or temporary variables.
  e.g. `i`, `j`
- Use consistent naming convention throughout one project.
- Don't write magic constants in a program. `#define` constants as a macro.

Layout and Indentation

- Use spaces and blank lines to make program easy to understand.
- Indent program according to its structure.

```
char line_buffer[81];             /* input line            */
int  buffer_index;                /* index into line_buffer */
int  c;                           /* input character       */
int read_one_line(void){
    c=getchar();
    while (c!='\n'){
        line_buffer[buffer_index++]=c;
        c=getchar();
    }
}
```

Debugging

- Don't write more than two statement in a single line (because debugger usually steps by line).

```
for (i=1; i<100; i++){c=getchar(); buf[i]=c;}
```

You cannot break inside this loop using debuggers.

- Don't optimize (by machine or by hand) unless your program is running correctly.
- Use `#ifdef DEBUG` ~ `#endif` to insert debugging statements (e.g. checking consistency of input data, etc.).

Size of a Function

- If a function size is too large (more than one page), consider breaking it into smaller functions.
- If a function has more than 7 local variables (if you cannot remember them all), consider breaking it into smaller functions.

Advices from "The Elements of Programming Style" (1)

"The Elements of Programming Style," B. W. Kernighan, 1979 is the classic book on Programming Style (the language discussed is FORTRAN and PL/I, but the advices in the book is applicable to all the languages)

Write clearly - don't be too clever.
Say what you mean, simply and directly.
Use library functions.
Avoid temporary variables.
Write clearly - don't sacrifice clarity for "efficiency."
Let the machine do the dirty work.
Replace repetitive expressions by calls to a common function.
Parenthesize to avoid ambiguity.
Choose variable names that won't be confused.
Avoid the Fortran arithmetic **IF**.
Avoid unnecessary branches.

Advices from "The Elements of Programming Style" (2)

Use the good features of a language; avoid the bad ones.
Don't use conditional branches as a substitute for a logical expression.
<u>Use the "telephone test" for readability.</u>
Use **DO-END** and indenting to delimit groups of statements.
Use **IF-ELSE** to emphasize that only one of two actions is to be performed.
Use **DO** and **DO-WHILE** to emphasize the presence of loops.
Make your programs read from top to bottom.
Use **IF…ELSE IF… ELSE IF… ELSE…** to implement multi-way branches.
Use the fundamental control flow constructs.
Write first in an easy-to-understand pseudo-language; then translate into whatever language you have to use.
Avoid **THEN-IF** and null **ELSE**.
Avoid **ELSE GOTO** and **ELSE RETURN.**
Follow each decision as closely as possible with its associated action.
Use data arrays to avoid repetitive control sequences.
Choose a data representation that makes the program simple.
<u>Don't stop with your first draft.</u>

Advices from "The Elements of Programming Style" (3)

Modularize. Use subroutines.
Make the coupling between modules visible.
Each module should do one thing well.
Make sure every module hides something.
Let the data structure the program.
Don't patch bad code – rewrite it.
Write and test a big program in small pieces.
Use recursive procedures for recursively-defined data structures.
Test input for validity and plausibility.
Make sure input cannot violate the limits of the program.
Terminate input by end-of-file or marker, not by count.
Identify bad input; recover if possible.
Treat end of file conditions in a uniform manner.

Advices from "The Elements of Programming Style" (4)

Make input easy to prepare and output self-explanatory.
Use uniform input formats.
Make input easy to proofread.
Use free-form input when possible.
Use self-identifying input. Allow defaults. Echo both on output.
Localize input and output in subroutines.
Make sure all variables are initialized before use.
<u>Don't stop at one bug.</u>
Use debugging compilers.
Initialize constants with **DATA** statements or **INITIAL** attributes; initialize variables with executable code.
Watch out for off-by-one errors.
Take care to branch the right way on equality.
Avoid multiple exits from loops.
Make sure your code "does nothing" gracefully.
Test programs at their boundary values.
<u>Program defensively.</u>

Advices from "The Elements of Programming Style" (5)

10.0 times 0.1 is hardly ever 1.0.
Don't compare floating point numbers just for equality.
Make it right before you make it faster.
Keep it right when you make it faster.
Make it clear before you make it faster.
Don't sacrifice clarity for small gains in "efficiency."
Let your compiler do the simple optimizations.
Don't strain to re-use code; reorganize instead.
Make sure special cases are truly special.
Keep it simple to make it faster.
Don't diddle code to make it faster – find a better algorithm.
Instrument your programs. Measure before making "efficiency" changes.
Make sure comments and code agree.
Don't just echo the code with comments – make every comment count.
Don't comment bad code – rewrite it.

Advices from "The Elements of Programming Style" (6)

Use variable names that mean something.
Use statement labels that mean something.
Format a program to help the reader understand it.
Indent to show the logical structure of a program.
Document your data layouts.
Don't over-comment.

IV. Writing Reliable Code

Contents

1. Common Mistakes

- How to avoid common mistakes
  - Obey coding rules (MISRA C, etc.)
  - Review and proofread programs
  - Turn on highest error checking level of the compiler
  - Use header files to keep consistency of the declarations in the program
  - Use program checker to detect mistakes

- Comments -

- Don't forget to close a comment.
  e.g.

```
/* Comment
      a++;
/* Comment */
      b++;
```

This statement is ignored.
No error message issued.

  – `//` comment (comment which ends at the end of line is available in C++ and C99, but not portable among traditional C compilers.

- False Indentation -

- Indent according to the program structure.

  e.g.

```
if (a==0)
    a++;
    b++;
```

This statement is outside the `if` statement.

  – Use `{}` to enclose sub-statements even if it consists of only one statement.

- Mistakes in conditions -

- Don't mistake `=` for `==` in conditions.

  e.g.

```
if (a=0){
    ...
}
```

Always false. 0 is assigned to `a`. No error message issued.

- De-referencing Uninitialized Pointers -

- Don't access using uninitialized pointers

  e.g.

```
f(){
  int *p;
  return *p;
}
```

Invalid memory area is accessed.

- Dereferencing `NULL` Pointers -

- Don't access through a `NULL` pointer.

e.g.

```
f(){
    int *p=NULL;
    *p=0;
}
```

Address 0 is accessed. System memory area might be clobbered.

- Using Uninitialized Variables -

- Initialize variable before it is used

  e.g.

```
f()
{
   int i;
   if (i==0){
      ...
   }
}
```

The value of `i` is undefied here

- Exceeding Array Bounds -

- Don't access an array with out-of-bound index.

  e.g.

```
char a[10];
f(){
   int i;
   for (i=0; i<=10; i++)
      a[i]=0;
}
```

`a[10]` is out of bound

- Forgetting `return` statement -

- Don't forget to write `return` statement of a function.

  e.g.

```
int sq(int x){
  int result;
  result=x*x;
}
void g(void){
  a=sq(10);
}
```

`return result;`
should be added.

- Cheating Types -

- Don't cheat types.
- Type cheating can be done in the following ways:
  - Through union members.
  - Through pointers (different type pointers pointing to the same area)
  - Through function parameters and return values

- Cheating Types (1) -

- ```
  union{
     long x;
     float y;
  } U;
  float f(){
     U.x=1;
     return U.y;
  }
  ```
  ```

- Cheating Types (2) -

- ```
  long *pl=(long *)100000;
  float *pf=(float *)100000;
  float f(){
    *pl=100;
    return *pf;
  }
  ```
  ```

- Cheating Types (3) -

file 1

```
float f(float x){
  ...
}
```

file 2

```
long f(long x);

g(){
  long x=f(100);
}
```

Endianness

- Cheating types causes serious problem when you port a program between machines with different "Endianness".
- Endianness determines how a word/long word is stored in memory.

Big Endian vs Little Endian

31			0
A	B	C	D

long word data on a register

Big Endian
68000, IBM, etc.

Memory

Little Endian
x86, etc.

Address 4n	A
4n+1	B
4n+2	C
4n+3	D

Address 4n	D
4n+1	C
4n+2	B
4n+3	A

Applications and Endianness

- x86 (Pentium, etc.) is little endian machine, so PC-related application requires little endian
- Network Protocol assumes big endian, so big endian machines are more efficient for network application
- SH is originally a big-endian machine. But SH3 and SH4 supports both endianness to support these application areas

Different Endian Gives Different Program Behavior

```
union{
  unsigned long l;
  unsigned char a[4];
} u;

...
  u.l=0x12345678;
  x=u.a[0];
```

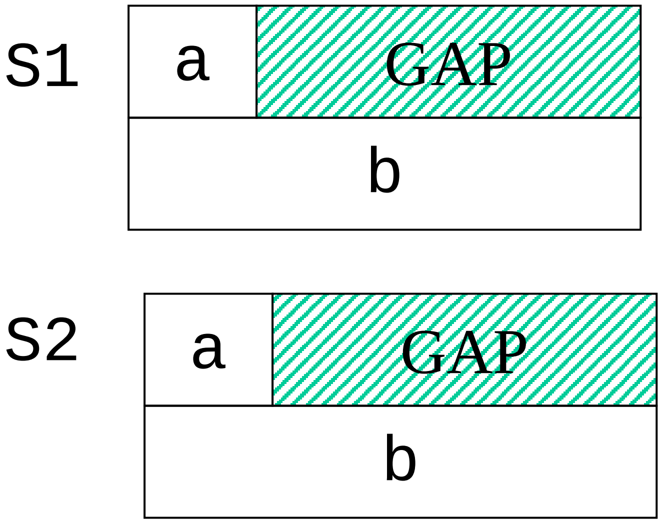
→ The value of `x` is 0x12 in Big Endian, 0x78 in Little Endian.

- Comparison of `struct`s as a Memory Areas -

- Structures have gaps between members.
- So comparing `struct`s as memory areas (using `memcmp` library functions, etc.) might not be correct.

```
struct{
    char a;
    int b;
} S1, S2;
```

The contents of gaps are not guaranteed



2. Assertion

- To develop a reliable program, make your program abort when anything wrong happens.
- Then fix all these problems.
- The macro `assert` generates error messages for specified error conditions during debugging, without any overhead in the final code.

Usage of `assert`

```
#include <assert.h>
double square_root(double x){
  assert(x>=0);
  /* Computation of square root */
  ...
}
```

insert
`#define NDEBUG`
here to disable assertions (when debugging is finished)

- `assert(`<expression>`)` is no-operation when macro `NDEBUG` is defined.
- Otherwise, if <expression> is false, prints message including the expression, filename, line number of the assertion (requires `printf`).

How to Use Assertions

- Assert function parameters
  - Assert assumptions for the function parameters.
- Assert input data
  - Assert to check the consistency of input data
- Assert possible error conditions
  - Assert array index is not out of bound
  - Assert that divisor is non-zero before division
  - Assert to check overflow

How `assert` works

```
#ifdef NDEBUG
  #define assert(cond) ((void)0)
#else
  #define assert(cond)\
    ((cond) ? ((void) 0) :
                printf("Assertion failed: %s, file: %s, line: %d\n",\
                       #cond, __FILE__, __LINE__),\
               abort())
#endif
```

`#cond` is replaced by a string out of a macro parameter cond.

`__FILE__`, `__LINE__` are replaced by the filename and line number of the macro call.

`((void) 0)` is a way to make the expression return `void` type

You can try similar construct when printf is not available in your system.

3. Unit Testing

- Unit testing is a testing method to test programs according to the program structure (white-box testing) (c.f. black-box testing, testing from outside the program).
- Sometimes black-box testing cannot test boundary conditions for each functions.
- The white box testing can be best documented if you include testing procedure in your program.
- Testing procedures should be maintained when you change the program.

Example

```c
#include <assert.h>
#define UT
int Max(int x, int y){
  if (x>=y) return x;
  return y;
}
#ifdef UT
void testMax(void){
  assert(Max(1, 0)==1);
  assert(Max(0, 1)==1);
  assert(Max(1, 1)==1);
}
#endif
int main(void){
#ifndef UT
  ...
#else
  testMax();
  /* Other tests */
#endif
}
```

Uses `assert` macro to help testing

Remove this line when testing is finished

Unit test procedure (designed to cover boundary cases)

Main processing for testing

Test Coverage

- C0 Coverage:
  - Coverage of Statements
    - No. of executed statements/No. of all the statements
- C1 Coverage:
  - Coverage of Branches
    - No. of branches executed/No. of all the branches
    - For each branch, both of taken one and not taken one are counted
- Make sure that you cover 100% of C0 and C1 in unit test
  - You must design test cases to cover these requirements
- Use coverage to measure the progress of system test and compare with number of bugs

4. Error Handling

- Embedded systems don't have "RESET" button. Error conditinos should be handled inside your program.
- Detect all the unexpected errors while debugging/testing.
- Design how to handle all the expected errors inside your program.

How to Handle Panic Situation

```
#include <setjmp.h>
jmp_buf init_env;
int main(void){
  int stat;
again:
  ... /* initialization */
  stat=setjmp(init_env);
  if (stat==0){
    ... /* main processing */
  }
  else{
    ... /* Do error processing */
    goto again;
  }
}
```

Initially, `setjmp` returns 0.
When longjmp is called, control returns here with `error_code` (nonzero) as return value.

```
#include <setjmp.h>
extern jmp_buf init_env;
f(){
  longjmp(init_env, error_code);
}
```

When error situation occurs, call longjmp with saved environment and error code.

5. Unsecure Library Usage

- `char *gets(char *buf)`
  - Reads a string from standard input to the buffer `buf`.
  - Buffer overflow occurs when input string exceeds the size of buffer.
  - The most serious problem is that, the buffer overflow depends on the input data, not on program logic.
  - Don't use this function, instead use `fgets` (which specifies buffer size)

Other security problems

- String libraries (`strlen`, `strcpy`, `strcat`, `strcmp`, etc.) assume that strings are terminated with null code.
- If null code is not found, memory area would be read/written indefinitely.
- When memory area is written indefinitely, other data would collapse.
- You should check input data (or parameters) before applying these functions.

V. Writing Efficient Code

RSO/Tools Marketing Dept.
Yugo Kashiwagi

Contents

1. Tuning-up Strategy

- First, reconsider the algorithm before tuning up.
- Measure the performance of the program to determine where to tune-up
- Add comment about what you have done. Retain the original code as a comment.
- Rely on compiler optimization whenever possible.

2. Data Structures
- Use 4-byte local variables -

```
int f(void)
{
  char a=10;
  int c=0;
  for (; a>0; a--)
  c+=;
  return(c);
}
```

→

```
int f(void)
{
  long a=10;
  int c=0;
  for (; a>0; a--)
  c += a;
  return(c);
}
```

Local variables/parameters are usually allocated to 4-byte registers. Declaring them as 4-byte data eliminates `EXTU/EXTS` instructions.

Specific to 4-byte CPUs

- Sign of Global Variables -

```
unsigned short a;
unsigned short b;
int c;
void f(void)
{
  c=b+a;
}
```

→

```
short a;
short b;
int c;
void f(void)
{
  c=b+a;
}
```

For 1/2 byte global data, prefer signed data type to unsigned data type. SH automatically sign-extends these data. Unsigned data requires EXTU instruction.

Specific to SH

- Put Related Data in a `struct` -

```
int a, b, c;
void f(void)
{
  a=1;
  b=2;
  c=3;
}
```

```
struct s{
  int a;
  int b;
  int c;
} s1;
void f(void)
{
  struct s *p=&s1;
  p->a=1;
  p->b=2;
  p->c=3;
}
```

Global variable requires 4-byte address to access. Structuring them reduces the usage of 4-byte addresses. This also improves data locality (better cache usage).

- Put Important Members at the Beginning of a `struct` -

```
struct{
{
  char buf[80];
  int key;
}
```

→

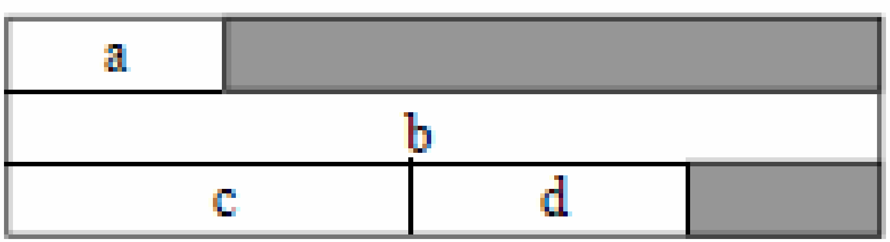
```
struct s{
  int key;
  char buf[80];
}
```

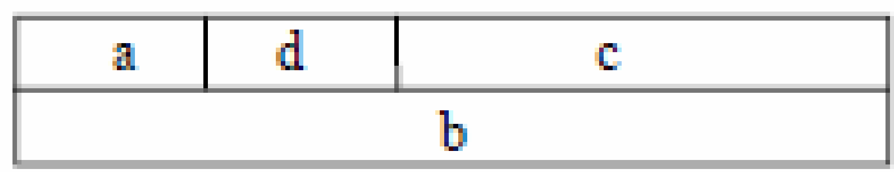
If the struct member is near the beginning, the offset to access the member is smaller, and more efficient code can be generated.

- Consider Data Alignment -

```
struct{
  char a;
  int b;
  short c;
  char d;
}
```

```
struct{
  char a;
  char d;
  short c;
  int b;
}
```





Reduce alignment gaps by declaring smaller data first.

- Use `const` Data -

```
char a[]={
  1, 2, 3, 4, 5
};
```

→

```
const char a[]={
  1, 2, 3, 4, 5
};
```

ROM data are less expensive than RAM data.
Declaration without `const` requires both RAM area and ROM area for initial values.

- Prefer Local Data to Global Data -

```
int i;
void f(void)
{
  for (i=0; i<10; i++)
    ;
}
```

→

```
void f(void)
{
  int i;
  for (i=0; i<10; i++)
    ;
}
```

Don't declare local data as global variables.
Local variables can be allocated on registers.

- Use Pointers to Access Array Elements -

```
int f1(int data[],
       int count)
{
  int ret=0, i;
  for (i=0; i<count;
i++)
    ret+=data[i]*i;
  return ret;
}
```

```
int f2(int *data,
       int count)
{
  int ret=0, i;
  for (i=0; i<count; i++)
    ret+=*data++ *i;
  return ret;
}
```

Using pointer may reduce the time of array element address calculation.

- Prefer Using Smaller Constants -

```
int i;
void f(void)
{
  i=0x10000;
}
```

```
int i;
void f(void)
{
  i=0x01;
}
```

Smaller constants require smalle code..

3. Function calls
- Put Related Functions in a File -

```
extern g(void);
int f(void)
{
  g();
}
```

→

```
int g(void)
{
}
int f(void)
{
  g();
}
```

Put related functions in a single file, so that compiler can optimize function call instruction (JSR -> BSR).

- Use Function Table instead of `switch` statement -

```
extern void A(void);
extern void B(void);
extern void C(void);
void f(int a)
{
  switch (a){
  case 0:
    A(); break;
  case 1:
    B(); break;
  case 2:
    C(); break;
  }
}
```

→

```
extern void A(void);
extern void B(void);
extern void C(void);
static int (*tbl[3])()={
  A, B, C};
void f(int a)
{
  (*tbl[a])();
}
```

`switch` statements has overhead of checking switch value. function table doesn't check input value.

- Pass a Pointer to `struct` instead of Many Parameters -

```
int f(int, int, int,
      int, int);
void g(void){
  f(1, 2, 3, 4, 5);
}
```

→

```
struct b{
  int a, b, c, d, e;
} b1={1, 2, 3, 4, 5};
int f(struct b *p);
void g(void)
{
  f(&b1);
}
```

Keep the number of parameters small so that all the parameters are passed through registers.
If not, consider passing parameters as a pointer to a `struct`.

- Macros vs Functions -

```
int abs(int x){
  return x>=0 ? x : -x;
}
f(){
  a=abs(b);
  c=abs(d);
}
```

```
#define abs(x) \
  ((x)>=0 ? (x) : -(x))
f(){
  a=abs(b);
  c=abs(c);
}
```

Macros don't have function call overhead.
But extensive use of macros make your program size very large.

C++ and C99 provides `inline` function declarations.

4. Operations - Pre-compute Constant Expressions in a Loop -

```
extern int a[100],
           b[100];
void f(void)
{
  int i, j;
  j=5;
  for (i=0; i<100; i++)
    a[i]=b[j];
}
```

→

```
extern int a[100],
           b[100];
void f(void)
{
  int i, j, tmp;
  j=5;
  tmp=b[j];
  for (i=0; i<100; i++)
    a[i]=tmp;
}
```

Pre-compute expressions which remains constant in the loop, before entering into the loop..

- Loop Unrolling -

```
extern int a[100];
void f(void)
{
  int i;
  for (i=0; i<100; i++)
    a[i]=0;
}
```

→

```
extern int a[100];
void f(void)
{
  int i;
  for (i=0; i<100; i+=2){
    a[i]=0;
    a[i+1]=0;
  }
}
```

Reduce the number of loops by unrolling
loops reduces the number of branch instructions.

- Use a table instead of (Simple) `switch` Statement -

```
int f(int i)
{
  int ch;
  switch (i)
  {
  case 0: ch='a'; break;
  case 1: ch='x'; break;
  case 2: ch='b'; break;
  }
}
```

```
char tbl[]={
  'a', 'x', 'b'
};
int f(int i){
  return (tbl[i]);
}
```

If a `switch` statement has a simple structure, consider using a table.

- Prefer comparison with zero -

```
int f(int x)
{
  if (x>=1)
    return 1;
  else
    return 0;
}
```

```
int f(int x)
{
  if (x>0)
    return 1;
  else
    return 0;
}
```

Usually, comparing with zero is expanded into simpler instructions.

- Put Error Processing in `else` Clause -

```
int x(int a)
{
  if (a==0)
    error_proc();
  else
    g(a);
}
```

→

```
int x(int a)
{
  if (a!=0)
    g(a);
  else
    error_proc();
}
```

Putting normal processing in `if` clause (instead of `else` clause), you can save one branch instruction in the normal processing. Don't sacrifice the speed of normal processing for error checking.

- Prefer `if` to small `switch` statement -

```
int x(int a)
{
  switch (a){
  case 1: a=2; break;
  case 10: a=4; break;
  default: a=0; break;
  }
}
```

→

```
int x(int a)
{
  if (a==1)
    a=2;
  else if (a==10)
    a=4;
  else a=0;
}
```

Using `if` statement reduce the overhead of input value check of switch statement.

5. Considerations of Cache and Pipeline
- Cache Consideration -

- Cache miss is a big penalty.
- Locate related funcitons and related data in small range (i.e. put them in a single file) to reduce cache misses.
- Reduce the number of random accesses in the important loop.
- Reduce the size of the innermost loop.

Exchanging Loop Variables

```
for (j=0; j<N; j++)
  for (i=0; i<M; i++)
    a[i][j]=b[i][j]+
            c[i][j];
```

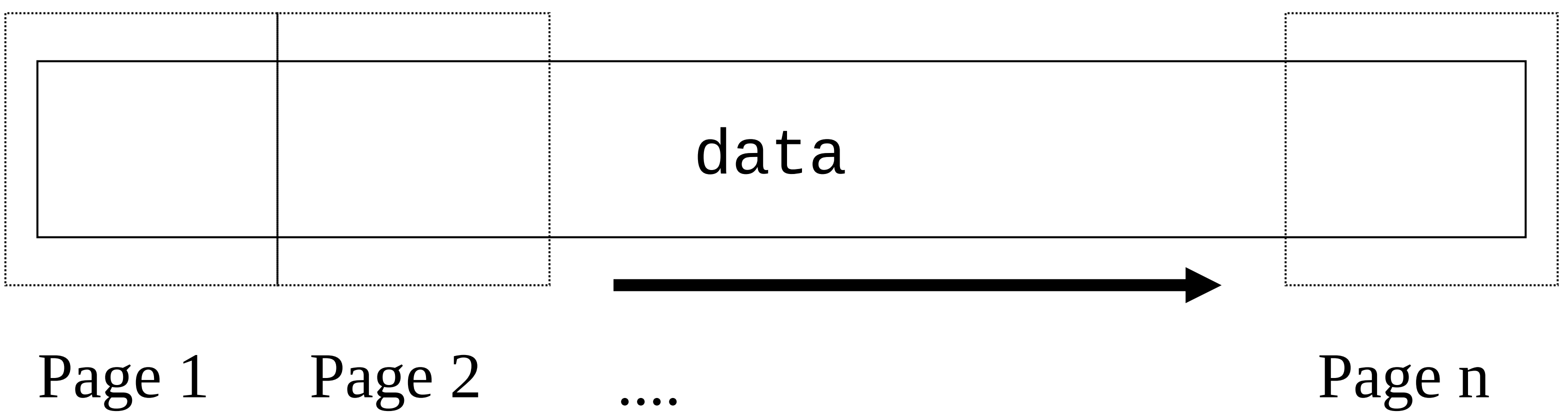
→

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    a[i][j]=b[i][j]+
            c[i][j];
```

Change the rightmost index in the inner loop, so that adjacent data are accessed in the innermost loop.
This reduces the number of cache misses.

Tiling

- When an array is too large to fit into a cache, and if you must travers the array many times, do the processing page by page.



Tiling

```
typedef struct {
  float a,b,c,d;
} data_t;
f(data_t data[], int n)
{
  data_t *p,*q;
  data_t *p_end = &data[n];
  data_t *q_end = p_end;
  float a,d;
  for (p = data; p < p_end; p++){
    a = p->a;
    d = 0.0f;
    for (q = data; q < q_end; q++){
      d += q->b -a;
    }
    p->d=d;
  }
}
```

This program computes the sum of difference of data[i].a and data[j].b (for all j) and stores into data[i].d.

```
#define STRIDE 512
f(data_t data[], int n)
{
  data_t *p,*q, *end=&data[n];
  data_t *pp, *qq;
  data_t *pp_end, *qq_end;
  float a,d;
  for (p = data; p < end; p = pp_end){
    pp_end = p + STRIDE;
    pp->d=0.0;
    for (q = data; q < end; q = qq_end){
      qq_end = q + STRIDE;
      for (pp = p; pp < pp_end &&
                   pp < end; pp++){
        a = pp->a;
        d = pp->d;
        for (qq = q; qq < qq_end &&
                     qq < end; qq++){
          d += qq->b -a;
        }
        p->d = d;
      }
    }
  }
}
```

Tiling

Before Tiling:

For each entry data[i], all the data[j] is scanned.

Data in cache changes n*(n/cache size) times.

After Tiling:

Inner two level loops runs without changing cache.

Data in cache changes (n/cache size)*(n/cache size) times.

Distribution of Accumulators

```
for (i=0; i<1000; i++)
  s+=a[i]*b[i];
```

```
s0=0;
s1=0;
s2=0;
s3=0;
for (i=0; i<1000; i+=4){
  s0+=a[i]*b[i];
  s1+=a[i+1]*b[i+1];
  s2+=a[i+2]*b[i+2];
  s3+=a[i+3]*b[i+3];
}
s=s0+s1+s2+s3;
```

In the old code, * waits until the array elements are loaded.
In the new code, they can run in parallel.

Software Pipelining

- If a loop contains a long operation (e.g. division or square root), you must wait until the operation is complete.
- Software pipelining is a technique to reconstruct the loop so that the long operation is started in the previous operation, and improve parallelism.

Software Pipelining

```
for (i=0; i<N; i++){
  x=X[i];
  y=Y[i];
  t=x/y;
  Z[i]=t;
}
```

```
x=X[0];
y=Y[0];
t=x/y;
for (i=1; i<N; i++){
  x=X[i];
  y=Y[i];
  Z[i-1]=t;
  t=x/y;
}
Z[i]=t;
```

Division waits data load, and store waits division.
If you fetch the data in the previous iteration, these operations can be executed in parallel.

IV. Hints on Numeric Computatoin

Contents

1. Terminology

- **Roundoff Error:** Error introduced when the result cannot be precisely represnted.
- **Truncation Error:** Error introduced when approximation algorithm is stopped in finite steps.
- **Loss of significant digits:** Loss of precision when you subtract two numbers near to each other.

```
 100001
- 99999
-------
      2
```

Terminology

- **Floating Point:** A representation of real number with sign, exponent and significant digits.
- **Fixed Point:** A representation of real number with fixed exponent assumed.
- **ULP (Unit in the Last Place):** The value of the least significant bit of a number. This depends on the value of the exponent.

Terminology

- **Overflow:** The situation when the absolute value of the result is too big for the representation.
- **Underflow:** The situation when the value of the result is too small (near zero) for the representation.
- **NaN (Not a Number):** The representation of the result of the operation, which is not defined (0/0, $\infty - \infty$, etc).

2. Floating Point Data Representation (IEEE)

Sign: 1 bit (0: +, 1: -)

Exponent: float: 8 bits (bias: 127)

double: 11 bits (bias: 1023)

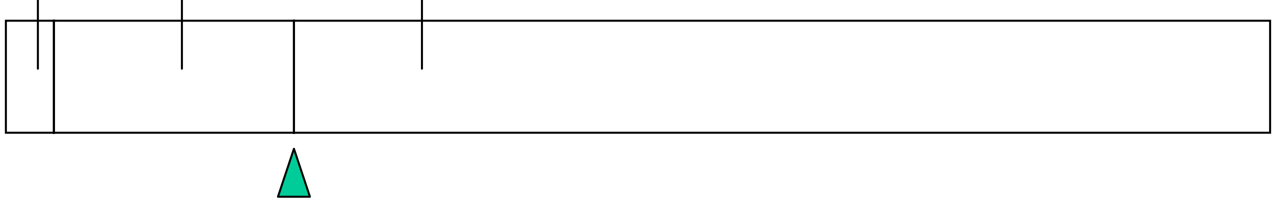
Mantissa: float: 23 bits, double: 52 bits,

MSB omitted

The represented value is:

$$(-1)^{\text{Sign}} * 2^{\text{Exponent-bias}} * (1+\text{Mantissa})$$

This bit is hidden before the point.



decimal (binary) point is assumed here

Floating Point Representation Example

- 0x3f800000
  - Sign: 0 (plus)
  - Exponent: 0x7f (127), represents 2^0
  - Mantissa: 0, represents 1.0
  - The represented value is +1.0
- 0x3f000000 -> +0.5
- 0x3fc00000 -> +1.5, etc.

Floating Point Representation (Special Values)

- Exponent=0, Mantissa=0: Zero
  - +0.0 and -0.0 are distinguished
- Exponent=max, Mantissa=0: Infinity
- Exponent=max, Mantissa!=0: NaN
  - Error value such as 0.0/0.0
- Exponent=0, Mantissa!=0:
  Denormalized number
  (represents very small values, with minimal exponent value. Mantissa doesn't assume hidden bit) (See SH4 Manual for details).

Arithmetic Operations and Rounding Modes

- IEEE Floating Point Arithmetic Operations (`+, -, *, /`) are defined as follows:
  - First compute "mathematically precise" result (you can find one as a rational number).
  - Then round the result according to rounding mode.
- There are several rounding modes (SH4 supports (1) and (4)):
  - (1) Round towards 0
  - (2) Round towards +infinity
  - (3) Round towards -infinity
  - (4) Round to nearest (default in most compilers)
    - When break even case, round so that the LSB becomes 0, i.e. round up if LSB is 1, and round down if LSB is 0.

3. Programming Hints
- General Principles -

- Consider the requirements of the accuracy.
- Avoid introducing errors.
- Always analyze errors.
- Consider algorithm. Don't directly apply formulas in your textbook.

- Prefer `float` to `double` -

- If the `float` is accurate enough for your application, use `float` type instead of `double`.
- Check your compiler's library if it has elementary functions for `float`. (In C99, use `sinf` instead of `sin`).
- Don't forget to add "`f`" postfix to your constants (e.g. `1.0f`), otherwise the arithmetic is done in `double` precision.

- Prefer Multiplication to Constant Division -

- Divison is slow.
- When dividing by constant, use multiplication with its reciplocal instead.
- e.g.
  `a=b/3.0` → `a=b*0.333333f`
- Note that the result is not exactly the same unless the divisor is a power of 2.0.

- Avoid Loss of Significant Digits -

- When subtracting (or adding values of opposite sign), consider the possibility of loss of significant digits.
- Use higher precision data or change algorithm to avoid such situations.

- Loss of Significant Digits (an Example) -

- Formula of quadratic equation: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- $\dfrac{-b - \sqrt{b^2 - 4ac}}{2a}$ is OK, but $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$ loses precision when *a* or *c* is small.
- Compute $\dfrac{-4ac}{2a\left(b + \sqrt{b^2 - 4ac}\right)}$ obtained by

  multiplying denominator and numberator by $b + \sqrt{b^2 - 4ac}$

- Sum up from Smaller Numbers -

- When summing up a series of numbers, add smaller number first.
- Consider $1.0+2^{-23}+2^{-23}$. 2^{-23} is very small and adding it to 1.0 results in 1.0. But if you add $2^{-23}+2^{-23}$, you get 2^{-22}, and adding it to 1.0 is not equal to 1.0.

- Computing Polynomials -

- Use Horner's Method to compute polynomials to reduce number of operations.

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

$$\downarrow$$

$$(\cdots((a_n x + a_{n-1})x + \cdots + a_1)x + a_0$$

4. Implementing Fixed Point Arithmetic

- If your CPU doesn't have FPU, and the dynamic range of the values are limited (-1.0~1.0, etc.), consider using fixed point.
- A sample implementation of 16-bit fixed point is presented (assuming that data range is -1.0~1.0)

Declaratoin and Constant Usage

```
typedef short FIXS16;

#define FIXS16_VAL(x) ((short)((x)*32768.0))

FIXS16 a=FIXS16_VAL(0.1234)
```

Computes corresponding `short` value at compilation time.

Addition and Subtraction

- Additions and subtractions are usual integer additions and subtractions.
- Make sure that the result doesn't overflow.

Multiplication and Division

```
FIXS16 MUL(FIXS16 x, FIXS16 y)
{
  return (x*y)>>15;
}
FIXS16 DIV(FIXS16 x, FIXS16 y)
{
  return (x<<15)/y;
}
```

VII. Compiler Optimizations

Register Allocation

- Compiler assigns registers to temporary expression results and some of the variables/constants.
- Number of registers are limited, frequently accessed variables (especially variables used in a loop) has higher priority.
- Variables in register generates more efficient code than variables in memory.

Expression Optimization

- Constant Floding
  - Computes constant expression at compile time

```
a=1+2;  →  a=3;
```

- Algebraic Simplification
  - Simplify expressions using algebraic equations

```
a= - b+c;  →  a=c-b;
```

Common Subexpression Elimination

- Common subexpression elimination detects same expression in the program, and avoids computing the expression more than once.

```
a=b+c;
x=b+c+d;
```

$\longrightarrow$

```
temp=b+c;
a=temp;
x=temp+d;
```

Constant Propagation

- Constant Propagation
  - Replace a variable by its assigned constant value.

```
limit=100;
for (i=0; i<limit; i++){
    ...
}
```

→

```
for (i=0; i<100; i++){
    ...
}
```

Copy Propagation

- Copy Propagation
  - Detect variables with the same value (by looking at assginments) and reduce the number of variables.

```
a=b;
x=a+10;
```

→

```
x=b+10;
```

Elimination of Redundant Statements

- Statements which are not executed, or has no effect is eliminated.

```
if (0)
  a++;
else
  b++;
```

→

```
b++;
```

```
f(){
  int a;
  a=1;
}
```

→

```
f(){
}
```

Loop Invariant Optimization

- Expressions which does not change value during a loop can be computed only once before the loop.

```
for (i=0; i<100; i++)
  a[i]=b+10;
```

→

```
temp=b+10;
for (i=0; i<100; i++)
  a[i]=temp;
```

Loop Unrolling

- Combines several iterations of a loop into one iteration to reduce number of branches.

```
for (i=0; i<100; i++)
  a[i]=0;
```

```
for (i=0; i<100; i+=2){
  a[i]=0;
  a[i+1]=0;
}
```

In-line Function Expansion

- Expands function directly in-line, instead of calling it.

```
int abs(x){
  if (x>=0)
    return x;
  else
    return -x;
}
f()
{
  y=abs(z);
}
```

→

```
f()
{
  int temp;
  temp=z;
  if (temp>=0)
    y=temp;
  else
    y=-temp;
}
```

Load/Store Optimization of Global Variables

- Cache a global variable in a register and reduce the number of load/store operation of the variable.
- Declaring a global variable `volatile` suppresses this optimization.

```
MOV @a,R1
MOV @a,R2
```

↓

```
MOV @a,R1
MOV R1,R2
```

```
MOV R1,@a
MOV @a,R2
```

↓

```
MOV R1,@a
MOV R1,R2
```

Pipeline Optimization

- Pipeline optimization reorders instructions so that pipeline stall is minimized.

```
MOV @R0,R1
ADD  #1,R1
MOV @R2,R3
ADD  #1,R3
```

→

```
MOV @R0,R1
MOV @R2,R3
ADD  #1,R1
ADD  #1,R3
```

When CPU tries to use the loaded data, it must wait until the data is actually loaded