Advance Programming in C

Yugo Kashiwagi

Microcomputer Tool Marketing Dept./RSO

Prerequisits and Objective

- Prerequisites
  - Basic programming skills in C
  - Some experience of program development
  - Familiarity with some CPU architecture
- Objectives
  - Basic knowledge of software engineering
  - Principles of algorithm design
  - Techniques of cache/pipeline optimization

Table of Contents

- Program Design
  - Structured Design
  - Modelling Techniques
- Algorithm Design
- Program Tuning
  - Assembler-level tuning
    - Pipeline
    - Cache

Program Design

- Structured Program Design
  - Abstraction and Encapsulation
  - Parametrization
  - Modular Program Design
    - Strength/Cohesion of Modules
    - Coupling of Modules
- Modelling
  - State Transition
  - Syntactic Modelling

Abstraction

- Abstraction gives an abstract description of a concrete representation (such as words and instructions).
  - Control Abstraction
    - Abstracts program structures
    - Main purpose of programming languages
  - Data Abstraction
    - Abstracts interface out of data representation
    - Supported by some high level languages (C++)
    - Should be considered while designing programs (independent of implementation langauge).

Control Abstraction

- Programming language is a tool to give an abstract description of concrete program and data.
  - Machine words and registers -> Vriables
  - Instruction sequence -> Procedures/Functions
  - Common pattern of branching -> control statements (if, while, for, etc.)

Data Abstraction

- Interface to a data structure should be independent from its inplementation.
  - You can improve the implementation of the data structure, without affecting its use.
  - You can develop the implementation of the data structure and its use in parallel.
  - Clear interface makes the program more maintainable.

Data Abstraction Example

Stack of integers

Interface:

```
void push(int x);
int pop(void);
int isEmpty(void);
```

usage:

```
f(){
  if (!isEmpty()){
    x=pop();
  }
  push(10);
}
```

Implementation:

```
#define MAXSTACK 100
int sp=0;
int stack[MAXSTACK];

void push(int x){
  stack[sp++]=x;
}

int pop(void){
  return stack[--sp];
}

int isEmpty(void){
  return sp==0;
}
```

Data Abstraction Example

Stack of integers: Alternative implementation (Same interface)

```
struct node{
  int data;
  struct node *next;
};

struct node *root=NULL;

void push(int x){
  struct node *p;
  p=malloc(sizeof(struct node));
  p->data=x;
  p->next=root;
  root=p;
}
```

```
int pop(void){
  int val;
  struct node *p;
  p=root;
  root=p->next;
  val=p->data;
  free(p);
  return val;
}

int isEmpty(void){
  return root==NULL;
}
```

Data Encapsulation

- Abstracting data is not enough. You should prohibit illegal access to the implementation of the data.
  - Use header files to share the interface among the modules.
  - Declare everything but the interface as "static"

Data Encapsulation Example

Interface: "stack.h"

```
extern void push(int x);
extern int pop(void);
extern int isEmpty(void);
```

usage:

```
#include "stack.h"
f(){
  if (!isEmpty()){
    x=pop();
  }
  push(10);
}
```

Declare concrete data and other auxiliary functions as "static"

Implementation:

```
#include "stack.h"
#define MAXSTACK 100
static int sp=0;
static int stack[MAXSTACK];

void push(int x){
  stack[sp++]=x;
}

int pop(void){
  return stack[--sp];
}

int isEmpty(void){
  return sp==0;
}
```

Parametrization

- Find common patterns in your program and implement it as a subroutine.
- When you copy-and-paste a segment of code, it is a good chance of parametrization.
- Parametrization make your code more compact and maintainable.
- Caution: Too much parametrization might complicate your code.

Parametrization Example

```
int a[100];

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

int b[1000];

void init_b(void){
  int i;
  for (i=0; i<1000; i++){
    b[i]=0;
  }
}
```

```
int a[100];
int b[1000];

void init_array(int *p, int size){
  int i;
  for (i=0; i<size; i++){
    p[i]=0;
  }
}
         :
         :
     init_array(a, 100);
     init_array(b, 1000);
```

More Sophisticated Parametrization Example

C Library Function "bsearch" searches an item from an array in which data elements are sorted. It is applicable to any size of data, any scheme of data comparison:

```
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *, const void *));
```

key: Pointer to a data to be sought.
base: Base address of the array.
nmemb: Number of elements in a table.
size: Size of the data.
compar: Comparison function of two data

Modular Program Design

- Modularity can be measured by two ways
  - Cohesion or Strength
    - Strength of relation inside a module
  - Coupling or Independence
    - Weakness of the relation among modules

Module Cohesion

- Coincidental Cohesion
- Logical Cohesion
- Procedural Cohesion
- Temporal Cohesion
- Communicational Cohesion
- Sequential Cohesion
- Functional Cohesion
- Informational Cohesion

Worst

Best

Module Cohesion

- Coincidental Cohesion
  - Routines have no relation at all.
- Logical Cohesion
  - Routines are collected, because they are in the same category, and selected by a "function code".
    - e.g. Input routine for various data, and the kind of data is selected by a "function code"

Module Cohesion

- Procedural Cohesion
  - Routines are collected, because they happen to be executed in a sequence (otherwise unrelated).
- Temporal Cohesion
  - Routines are collected, because they are done at the same time.
    - e.g. Initialization, shut down process, etc.

Module Cohesion

- Communicational Cohesion
  - Routines are collected, because they happen to access to the same data
    - Various operation to the same table
- Sequential Cohesion
  - Routines are collected, because the output of a routine becomes an input of another
    - e.g. Read_data, process_data, write_data.

Module Cohesion

- Functional Cohesion
  - None of the above
  - The module have well-defined clear description
- Informational Cohesion
  - Collection of functional-cohesion modules under well-defined description
    - e.g. Mathematical function library (sin, cos, tan, etc), Access modules to a single abstract data.

Module Coupling

- Content Coupling
- Common Coupling
- External Coupling
- Stamp Coupling
- Data Coupling
- No direct Coupling

Worst

Best

Module Coupling

- Content Coupling
  - One module modifies or relies on other module's internals
- Common Coupling
  - Two modules share (unrestricted) common global data
- External Coupling
  - Two modules share some specific common global data (interface is well-defined)

Module Coupling

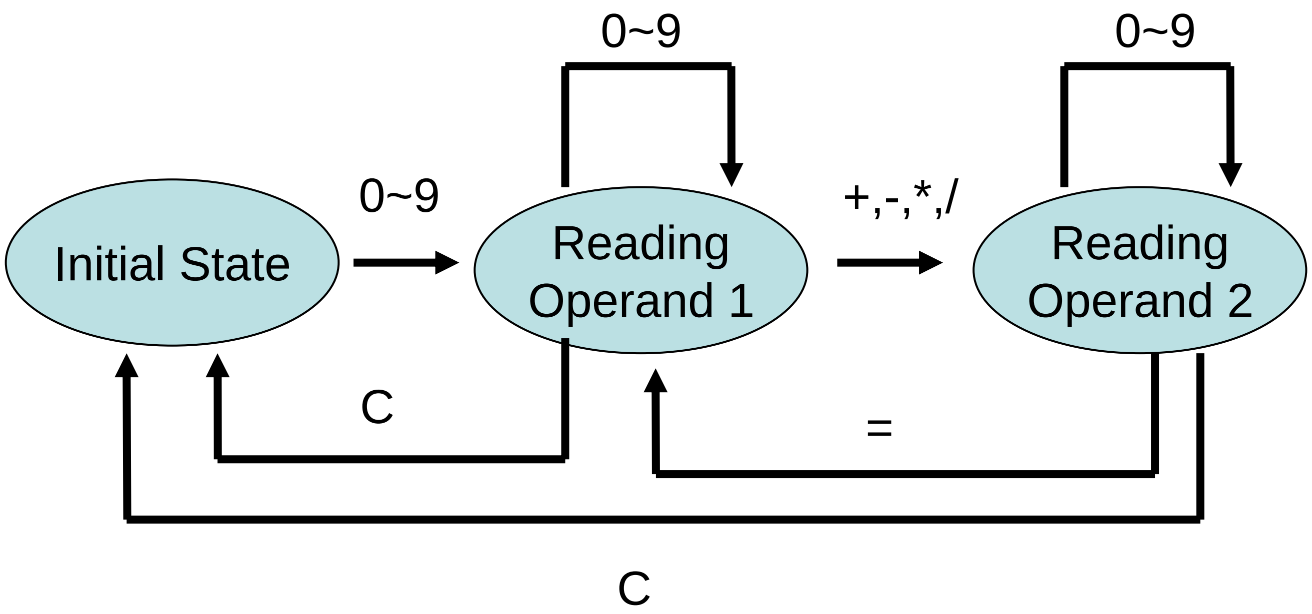
- Control Coupling
  - One module passes "function code" to another module
- Stamp Coupling
  - Passes and returns structured data
    - Changing data structure affects another module
- Data coupling
  - Passes and returns only simple data
- No direct coupling
  - None of the above

Modelling Techniques

- It is useful to model your problem in some abstract manner before you start the program design.

Modelling Technique: State Chart

Desk Calculator Example:



Program from State Chart

```
c=getchar();
switch (state){
case INIT:
  if (isdigit(c)){
    mem=c-'0';
    state=READ1;
  }
  break;
case READ1:
  if (isdigit(c)){
    mem=mem*10+c-'0';
  }
  else if (c=='C'){
    mem=0;
    state=INIT;
  }
  else if (c=='+' || ...){
    op=c;
    operand=0;
    state=READ2;
  }
  break;
case READ2:
  if (isdigit(c)){
    operand=operand*10+c-'0';
  }
  else if (c=='C'){
    mem=0;
    state=INIT;
  }
  else if (c=='='){
    mem=compute(op, mem, operand);
    printf("%d¥n", mem);
    state=READ1;
  }
  break;
}
```

Syntactic Modelling

- Programming languages are described by a grammar called BNF (Backus-Naur Form).
  - Example:
    ```
    <block>::="{" <declaration>* <statement>* "}"
    <if statement>::="if" "(" <expression> ")"
                        <statement>
                        ["else" <statement>]

    <item>*: 0 or more items
    <item>+: 1 or more items
    [<item>]: optional item
    <item1>|<item2>: one of <item1> or <item 2>
    ```
- You can formalize your input data using BNF and structure your program.

Modelling using BNF

<input>::=<header> <data record>* <trailer>

Suppose the header includes the number of data records.

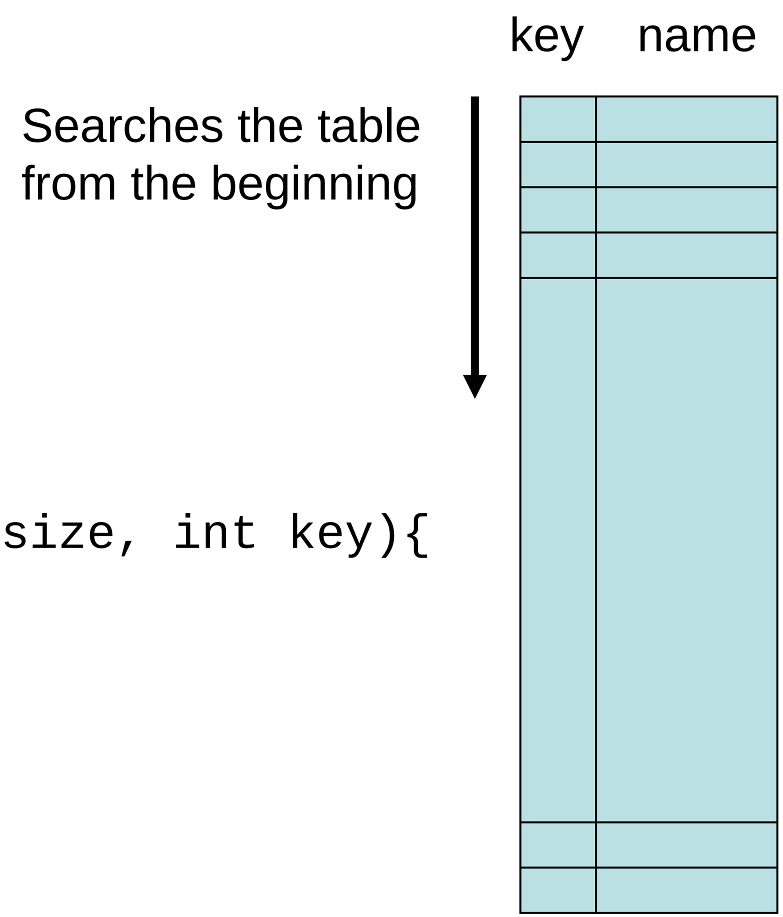
```
count=read_header();    /* This returns number of records */
for (i=0; i<count; i++){
  read_data_record();
}
read_trailer();
...
```

Algorithm Design
O notation

- O (big-O) notation
  - The execution time of an algorithm is a function of its input size. The order of this function (ignoring constant factor) is expressed using O notation.
    - O(n): Linear time
    - O(n log n): Frequently appears in many algorithms, sorting, FFT
    - $O(n^2)$: Impractical for large inputs
    - $O(n^3)$: e.g. Matrix Multiplication
    - $O(e^n)$: Impractival even for medium sized inputs
  - Constant factor is sometimes important (too large constant factor makes some algorithms impractical).

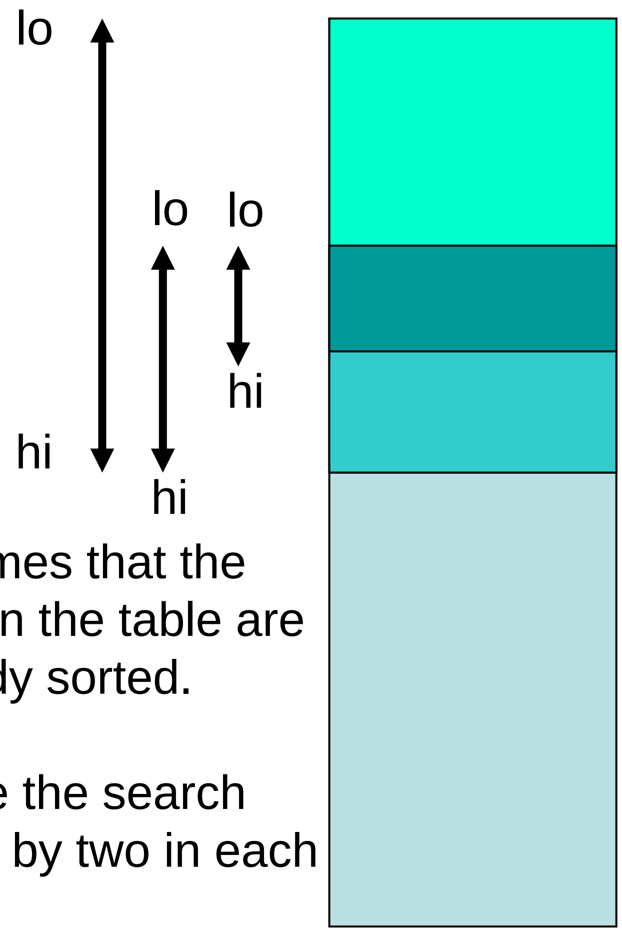
Example: Searching
Linear Search (O(n))

```
#include <stdio.h>

struct record{
  int key;
  char *name;
};

char *search(struct record *a, int size, int key){
  int i;
  for (i=0; i<size; i++){
    if (key==a[i].key){
      return a[i].name;
    }
  }
  return NULL;
}
```



Binary Search (O(log n))

```
char *search(struct record *a, int size, int key){
  int lo, hi, mid;
  lo=0;
  hi=size-1;
  while (lo<=hi){
    mid=(lo+hi)/2;
    if (a[mid].key<key){
      lo=mid+1;
    }
    else if (a[mid].key>key){
      hi=mid-1;
    }
    else{
      return a[mid].name;
    }
  }
  return NULL;
}
```



Assumes that the keys in the table are already sorted.

Divide the search range by two in each step.

Simple Insertion Sort ($O(n^2)$)

```
/* Sorts the index range [start..end] of array "a" */
void sort(int *a, int start, int end){
  int i, j, k;
  int tmp;
  for (i=start+1; i<=end; i++){
    /* At each step, the range [start..i-1] is */
    /* already sorted                          */
    tmp=a[i];
    j=i-1;
    /* Insert a[i] at appropriate place.       */
    while (j>=start && tmp<a[j]){
      a[j+1]=a[j];
      j--;
    }
    a[j+1]=tmp;
  }
}
```

Quicksort (O(n log n) (in average)

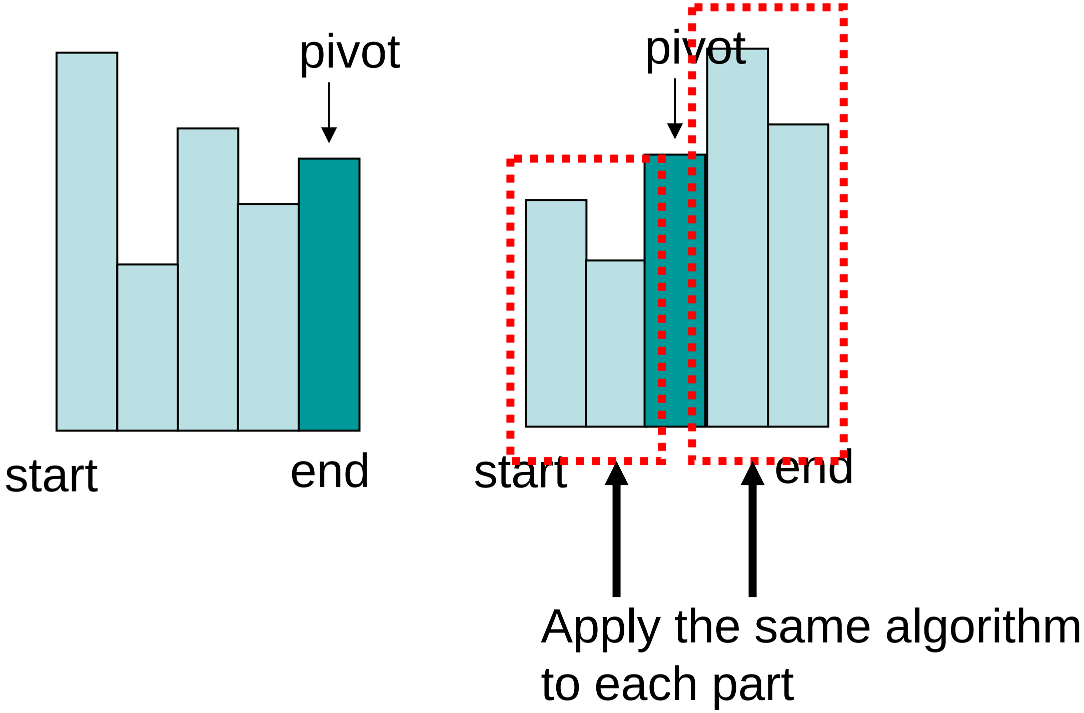
```
void sort(int *a, int start, int end){
  int i, j;
  int pivot;
  int tmp;
  if (start+1<end){
    pivot=a[end];
    i=start;
    j=end-1;
    while (i<j){
      while (i<j && a[i]<=pivot){
       i++;
      }
      while (i<j && a[j]>=pivot){
       j--;
      }
```

```
        if (i<j){
         tmp=a[j];
         a[j]=a[i];
         a[i]=tmp;
         i++;
         j--;
        }
      }
      a[end]=a[i];
      a[i]=pivot;
      sort(a, start, i-1);
      sort(a, i+1, end);
    }
  }
```

Quicksort (Explanation)

- Pick up one element (pivot) in the table (a[end]), and arrange the array so that elements before the pivot is smaller than the pivot, and those after the pivot is larger than the pivot. Thus the table is divided into two parts, in which the lower half has smaller elements, and the upper half has larger elements.
- Apply the same algorithm to both part of the table.
- Each step takes time proportional to n.
- The expected number of steps are log n.

Quicksort (Explanation)



Problem of the Quicksort (1)

- Worst case: If the partition is not even, the execution time can be $O(n^2)$.
  (In the example program, the worst case happens when the data is already sorted).
  - Select the pivot randomly, or
  - Select several pivots, and take medium sized one.

Problem of the Quicksort (2)

- Recursive calls wastes the stack area.
- In the worst case, the recursion depth is proportional to the table size n (impractical for large n).
  - Transform the function, and remove the tail recursion (there is still one recursive call left).
  - Do the smaller part first, so that recursion cannot be so deep (less than $\log_2 n$)

FAIL

Improvement of Quicksort

```
void sort(int *a, int start, int end){
  int i, j;
  int pivot;
  int tmp;
  while (start+1<end){
    pivot=a[end];
    i=start;
    j=end-1;
    while (i<j){
      while (i<j && a[i]<=pivot){
        i++;
      }
      while (i<j && a[j]>=pivot){
        j--;
      }
      if (i<j){
        tmp=a[j];
        a[j]=a[i];
        a[i]=tmp;
        i++;
        j--;
      }
    }
    a[end]=a[i];
    a[i]=pivot;
    if (i-start>end-i){
      sort(a, i+1, end);
      end=i-1;
    }
    else{
      sort(a, start, i-1);
      start=i+1;
    }
  }
}
```

Hints on Algorithm Design

- Divide and Conquer: Divide the problem into smaller sized subproblems (e.g. Quicksort).
- Recursive Design: You may be able to apply the same algorithm to the subproblem (but try to remove recursion when implementing).
- Consider the order of execution time.
- Sophisticated algorithms are hard to invent. Consult textbooks.
- Don't apply the textbook algorithm as it is. Consider the assumption of your own algorithm (data structure, imput assumption, etc.)

Programming Exercise Project 1

1. Run the quicksort and exchange sort for arrays of various size and measure execution cycles.
2. Remove the recursion of the quicksort and do the same thing. unresolve
3. [Bonus Problem] For smaller tables, exchange sort should run faster. Change the quicksort so that small-sized range is sorted using exchange sort. Which value is the optimal? exchange sort

with large input, quick sort takes so much time

????

Hew Operation Demo

- Building and running programs
- Counting machine cycles
- Using trace to observe pipeline status

Report

- Organization of the report
  - Give the summary and conclusion in the first page, then show details.
    - Summary includes:
      - Cycle counts before and after improvement (table or preferabry a graph)
      - Show what improvement you have done
      - Compare your expectation (based on what you have learnt) and the result.
    - Details:
      - Details of the program code (where you have changed)
      - Detailed observation and discussion
      - Questinos or suggestions
- Scoring policy
  - If your summary includes the required items, you get 70 pts.
  - Good discussion, question, suggestion gives extra pts.

Program Tune-up

- Introduction to Assembler
- Pipeline
- Cache
- Programming Exercise

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.
- <u>We recommend you to first understand what is happening with assembler programs, and then use non-portable features of each compiler.</u>

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

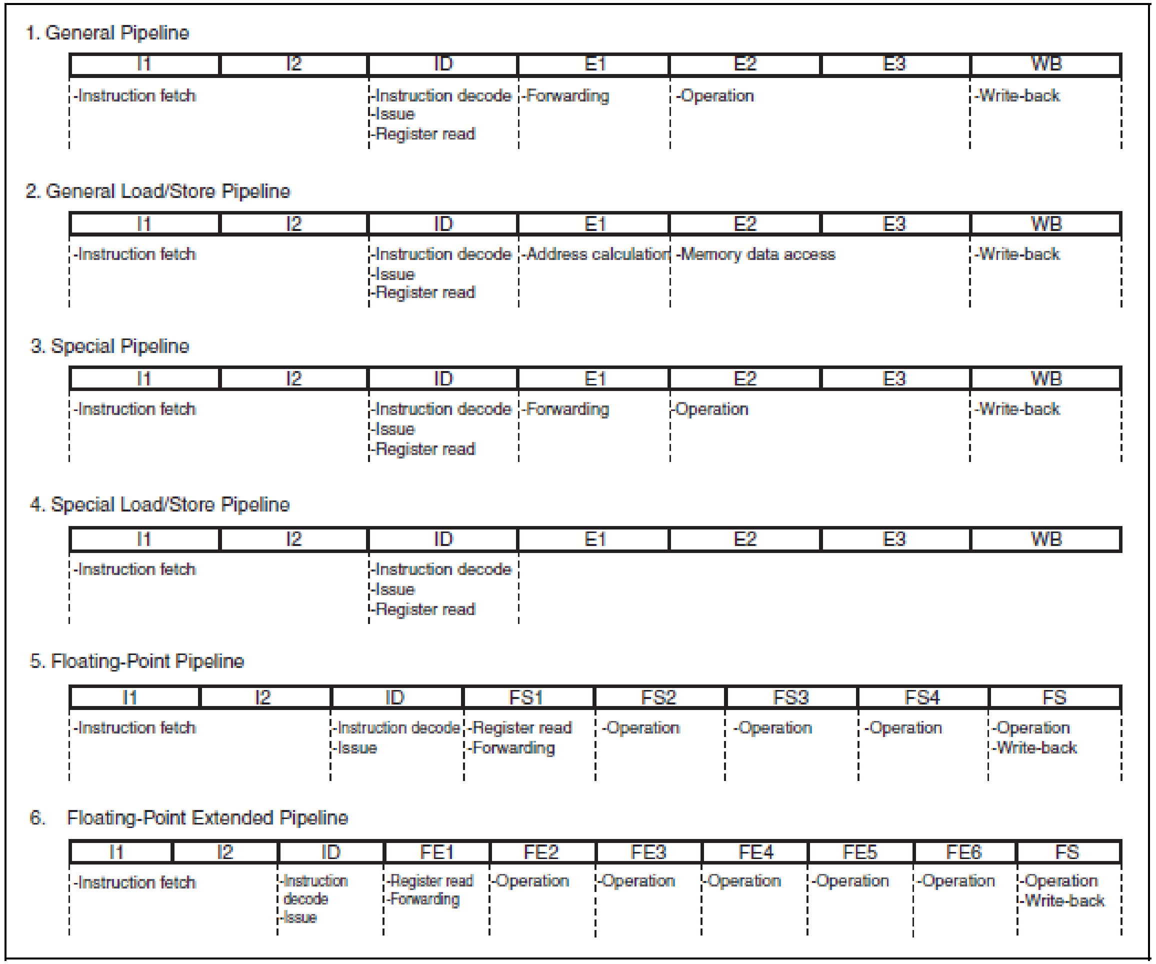
Function example (Register saving/restoring)

```
_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).

SH4A Pipeline Structure

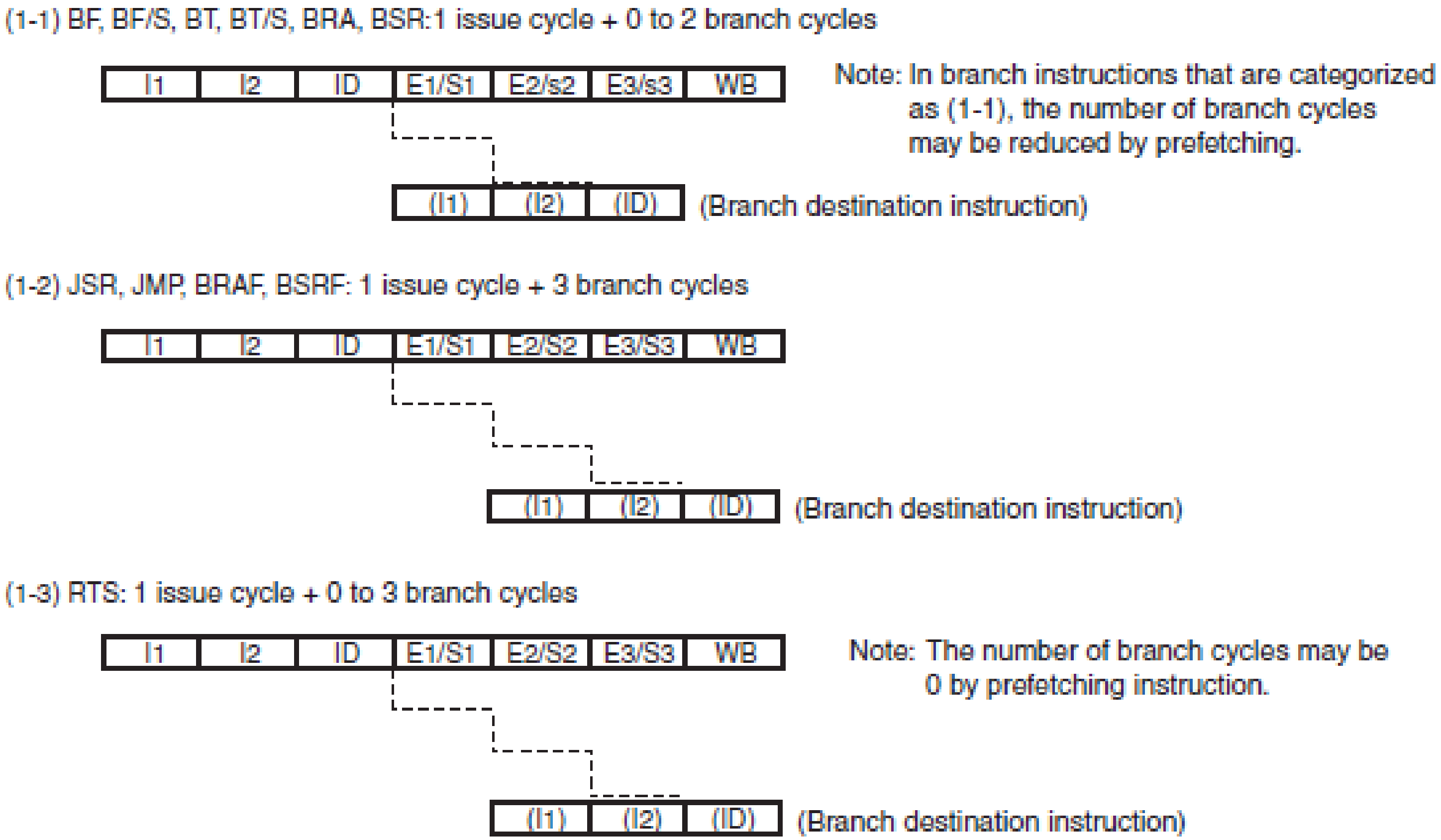


7 Stage Pipeline

I1/I2: Instruction Fetch
ID: Instruction Decode
E1-E3: Execution
WB: Write Back

Each stage is executed in parallel in different unit. So each instruction is executed in one cycle.

Pipeline: Branch Penalty



Branch requires the instructions to be fetched from another place.
So there are 2-3 cycles before next instruction can be executed

-> Use delay slot.

-> Reduce the number of branches (loop unrolling, etc.)

Pipeline: Superscalar

SH4A is a superscalar architecture, and can issue two instructions at a time.

However, same type instructions (except MT) cannot be done in parallel (because they use the same resources).

-> Interleave arithmetics and moves

		Preceding Instruction (addr)					
		EX	**MT**	**BR**	**LS**	**FE**	**CO**
Following Instruction (addr+2)	EX	No	Yes	Yes	Yes	Yes	
	MT	Yes	Yes	Yes	Yes	Yes	
	BR	Yes	Yes	No	Yes	Yes	
	LS	Yes	Yes	Yes	No	Yes	
	FE	Yes	Yes	Yes	Yes	No	
	CO						No

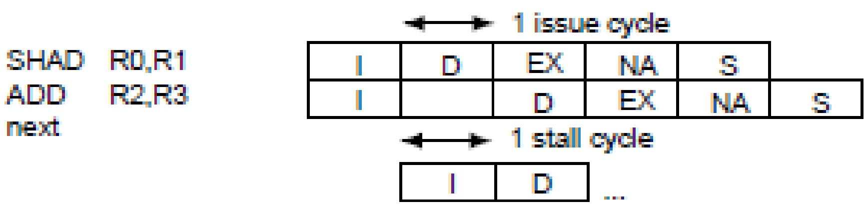
EX: Arithmetic
MT: Move (R-R or immediate-R)
BR: Branch

LS: Load/store
FE: Floating Point
CO: Control

Pipeline: Pipeline Stall

- SH4A Software Manual doesn't have examples. Here examples from SH4 (5-stage pipline) are shown to explain ideas.

(a) Serial execution: non-parallel-executable instructions



EX-group SHAD and EX-group ADD cannot be executed in parallel. Therefore, SHAD is issued first, and the following ADD is recombined with the next instruction.

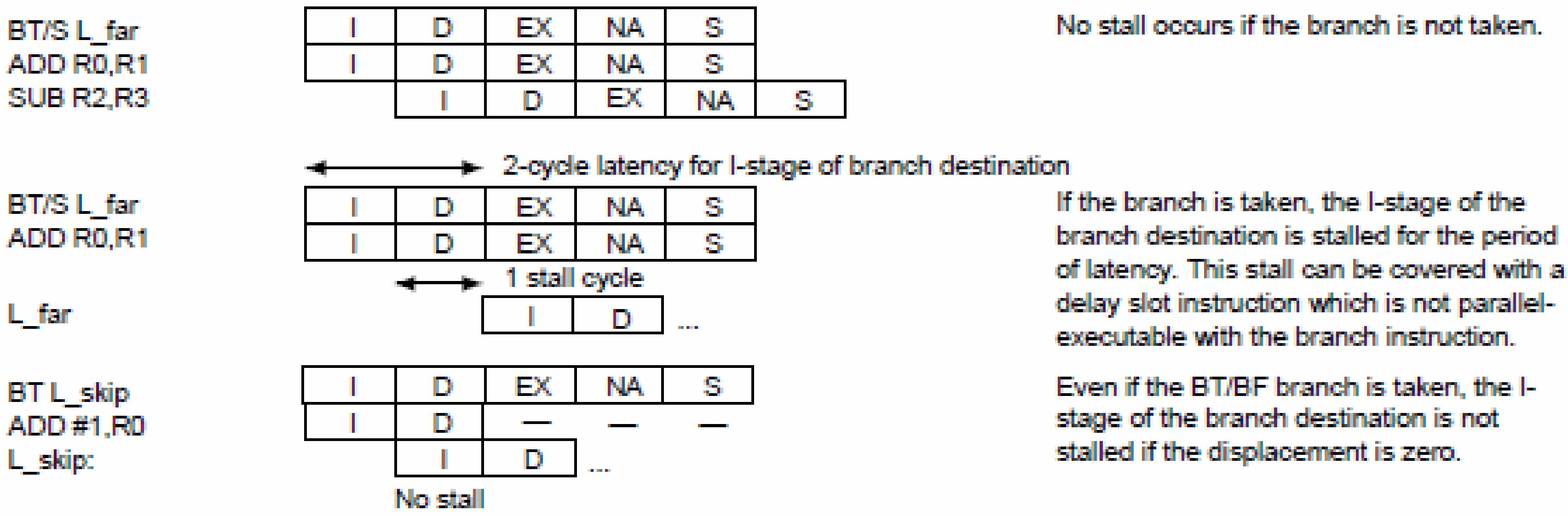
(b) Parallel execution: parallel-executable and no dependency

EX-group ADD and LS-group MOV.L can be executed in parallel. Overlapping of stages in the 2nd instruction is possible.

Serial/Parallel execution of superscalar. The instructions of the same group cannot be executed in parallel.

Pipeline: Pipeline Stall

(d) Branch



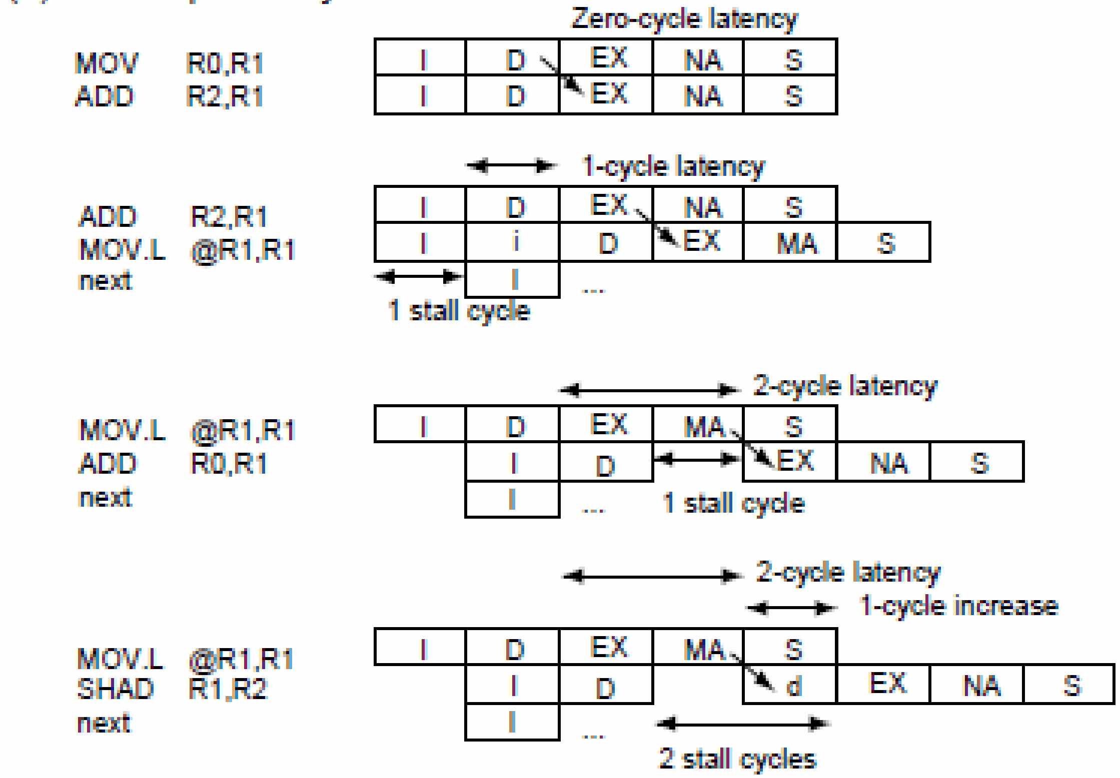
No pipeline stall if the branch is not taken

-> Design the program so that frequent path takes less branches

(e.g. then-part of if includes frequent processing)

Pipeline: Pipeline Stall

(e) Flow dependency



The following instruction, ADD, is not stalled when executed after an instruction with zero-cycle latency, even if there is dependency.

ADD and MOV.L are not executed in parallel, since MOV.L references the result of ADD as its destination address.

Because MOV.L and ADD are not fetched simultaneously in this example, ADD is stalled for only 1 cycle even though the latency of MOV.L is 2 cycles.

Due to the flow dependency between the load and the SHAD/SHLD shift amount, the latency of the load is increased to 3 cycles.

Don't use register immediately after it is operated or loaded (except register-register move).

Pipeline Consideration in C Programming

- How can we improve pipeline performance in C programs?
  - Reduce the number of "taken" branches
  - Improve instruction-level parallelism (don't make the loop too "tight", so that the compiler can find several things which can be done in parallel.

- Macros vs Functions -

Macros and inline functions reduces number of branches

```
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.

- 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<50; i+=2){
    a[i]=0;
    a[i+1]=0;
  }
}
```

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

This reduces the number of "taken" branches.

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;
```

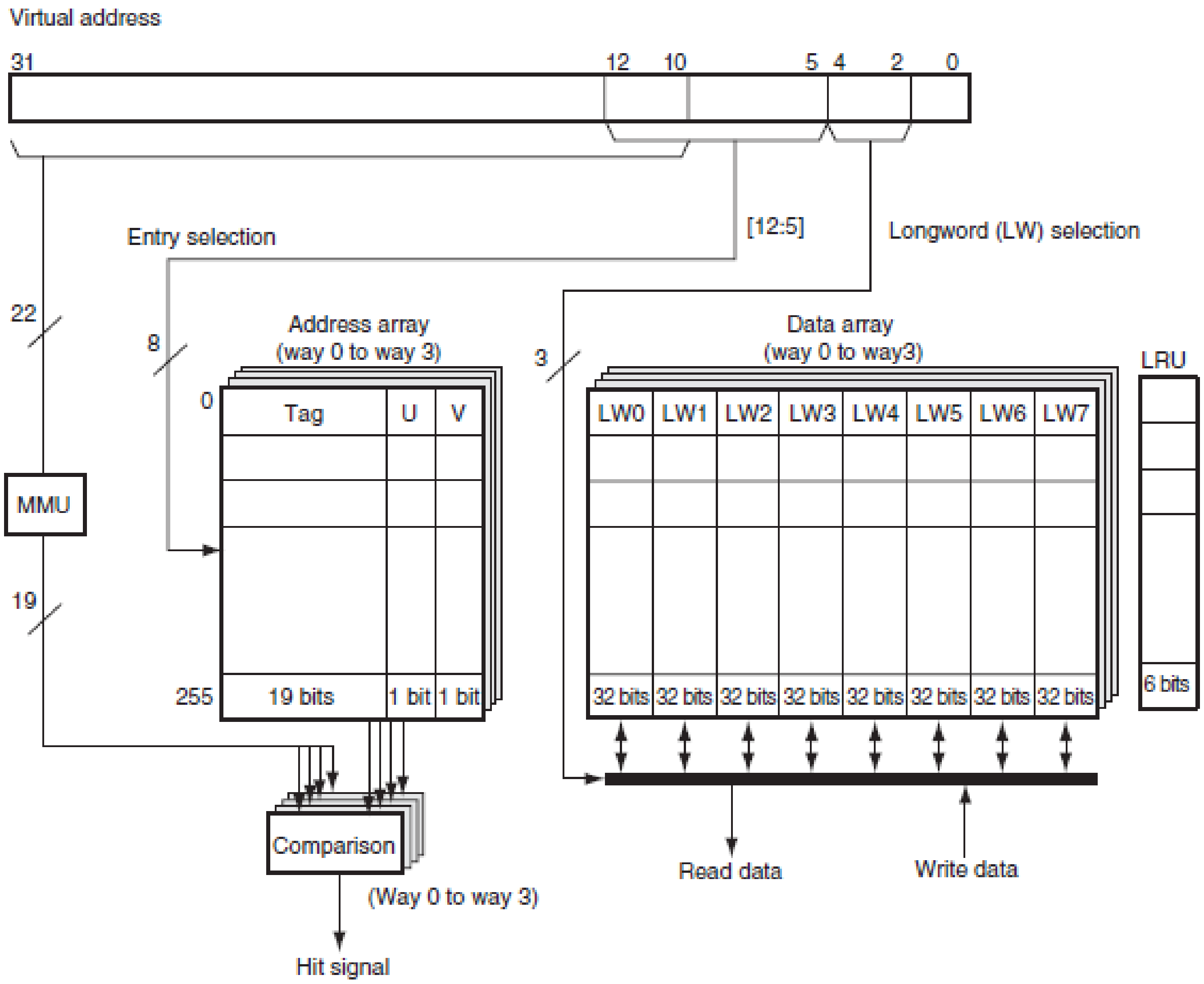
Division waits data load, and store waits division.
If you fetch the data in the previous iteration, these operations can be executed in parallel.

Cache Optimization

- Cache is a small but fast memory, which buffers the memory.
- Programs run fast when memory access hits the cache. Otherwise slow memory access with large penalty occurs.

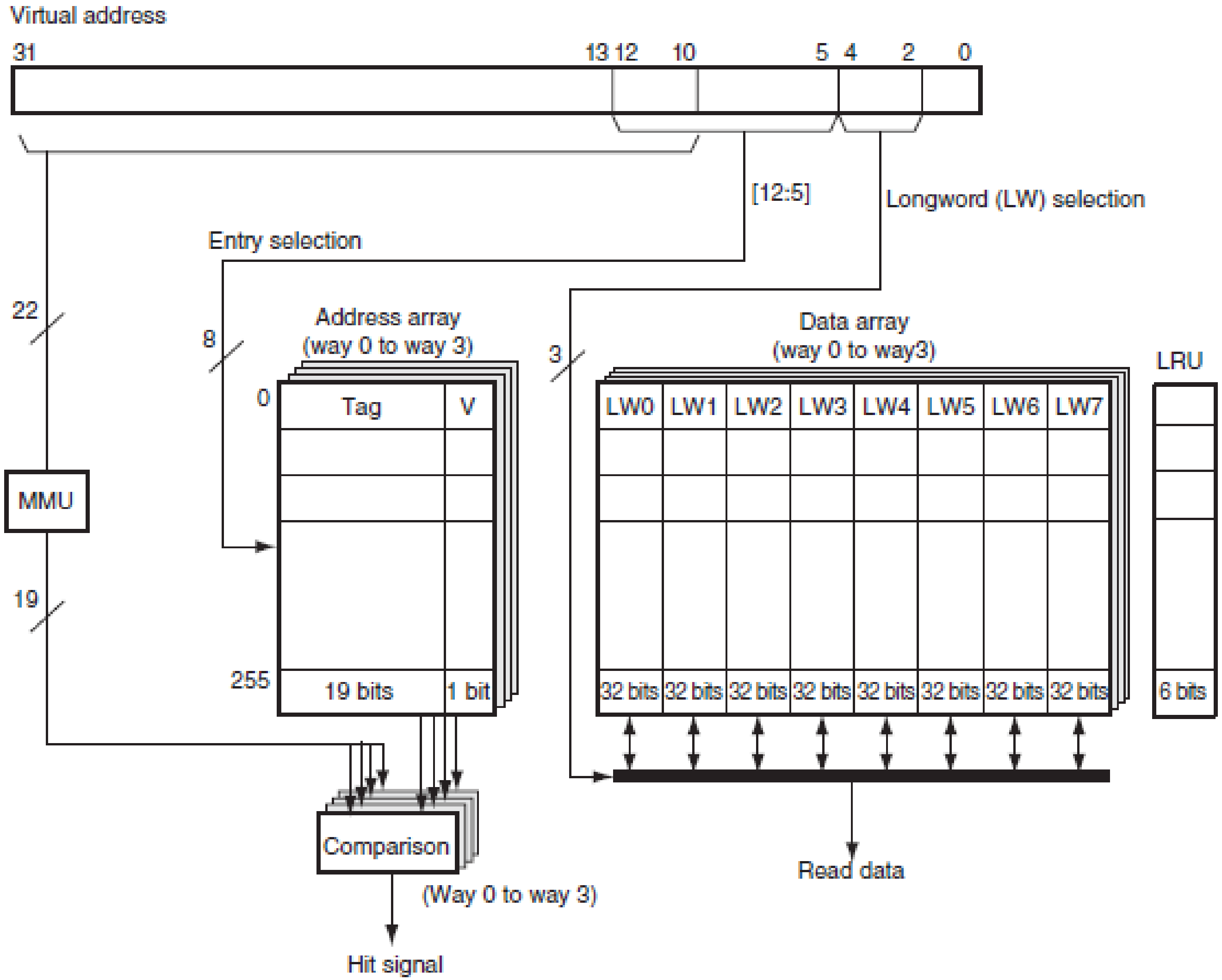
Cache

SH4A Operand Cache: 32K bytes, 4-way set associative, Line size=32



Cache

SH4A Instruction Cache: 32K bytes, 4-way set associative, Line size=32 (similar to operand cache)



Cache: Things to consider

- Line size (32 byte) is a unit to be loaded into the cache.
- 256 entries in each way can hold 256*32 (=8K) bytes of consecutive memory area.
- Up to 4 lines of the same address (modulo 8K bytes) can be kept in cache. If another data with the same address (modulo 8K) is accessed, the least recently used line is purged.

Cache: Basic Techniques

- Align data and program to 32-byte (line size) boundary
- Improve data/program locality
- Prefer sequential data access to random access
- Use prefetch (instruction or data, implemented as a intrinsic function in Renesas C) some time before data is needed (or function is called).

Exchanging Loop Variables

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

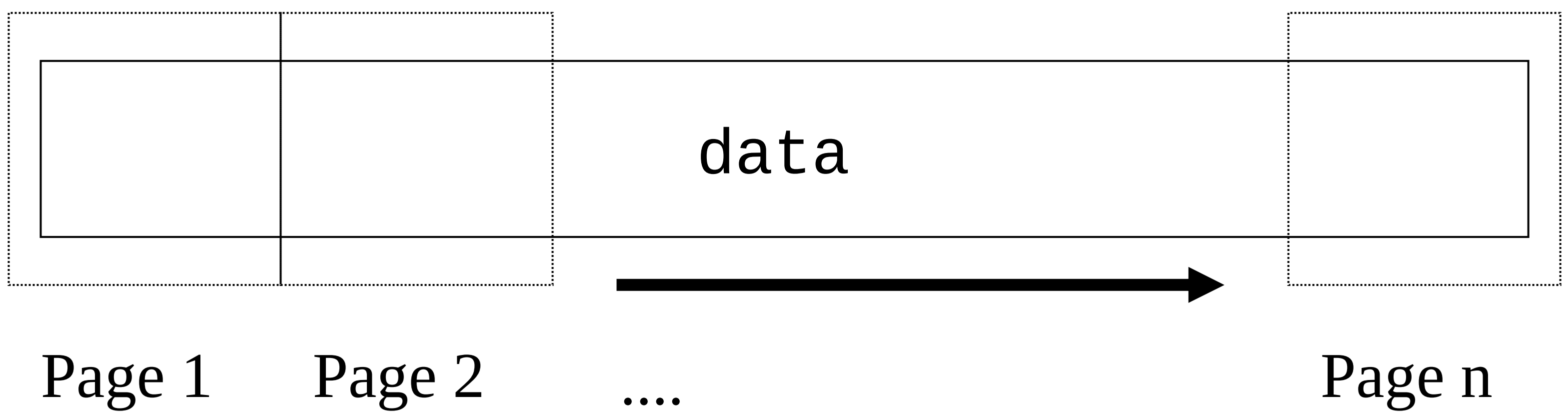
→

```
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.

Cache Prefetching (data)

```
#include <machine.h>   /* Renesas header for intrinsic instructions */

int a[2096];

   prefetch(a);
     :
   for (i=0; i<2096; i+=8){
     prefetch(&a+i+8); /* Prefetch next set of memory data.          */
     sum+=a[i];
     sum+=a[i+1];
     sum+=a[i+2];
       :
     sum+=a[i+7];
   }
```

Cache Prefetching (instruction)

```
#include <machine.h>   /* Renesas header for intrinsic instructions */

int a[2096];

   prefi(&f);
     :
   f();
     :
```

Programming Exercise Project 2

- The program sums up the elements of big (larger than cache size: 32K Byte) array.

```
int a[ARRAY_SIZE];

void f(void){
  int i;
  long sum;
  sum=0;
  for (i=0; i<ARRAY_SIZE; i++){
    sum+=a[i];
  }
}
```

Programming Exercise
Project 2

1. Apply "loop unrolling". Unroll it by 2, 4, 8 and measure their execution cycles.
2. Apply "distribution of accumulators". Try 2, 4, 8 accumulators and measure the execution cycles.
3. Using the best result from 1 and 2, insert "prefetch" for cache optimization.
4. [Bonus Problem] Study compiler-generated assembly program. Try more optimization in assembler level.

Requests for Contribution

- This is the first trial of "Advanced Programming Course"
- I'd like to include more examples. When you solve a hard programming problem, please write the summary and send it to me.
- Any volunteer teaching assistant next time?