Chapter 1 (1)

# Advanced C Techniques for System Programming

Kansas State University

# Advanced C Techniques for Embedded Systems Programming

- Basic Computer Architecture

- Execution Environment of C program

# Compiler, Assembler, Linker

- Compiler, Assembler, Linker

  Source File (.c) → Assembly File (.s) → Object File (.o) → Executable File (a.out)
  gcc -S file.c        gcc -c file.s        gcc file.o

  — An assembler is a program specialized for translating symbolic programs (text) into machine language programs (binary)

- Most old operating systems and many old application programs were written in assembly languages
  — Assembly programs are more efficient (time/space)
    – e.g., 8088 was not fast enough
  — There was no good high-level language for the purpose
    – e.g., systems programming
  — Architecture models were not suited to high level languages
    – e.g., old 8-bit machines (z80, 6800, etc) did not have enough registers to run high level languages efficiently

# Compiler, Assembler, Linker (cont)

- Now, almost all commercial programs are written in C (and VB, .NET, and Java)
- In operating systems and embedded systems, only  few percent is written in assembly code, and the rest are written in C
- If you develop/port operating systems, and/or develop embedded systems, it is essential to grasp the run time behavior of your code at the machine instruction level
  - execution environments, run-time memory allocation, etc.
  - techniques to link assembly code to C programs, and vice versa
- Even though you develop only application programs, it is important to understand the run-time behavior of your program
  - You will have less errors
  - When you have bugs, it is much easier to find and correct the errors.

# The von Neumann Machine

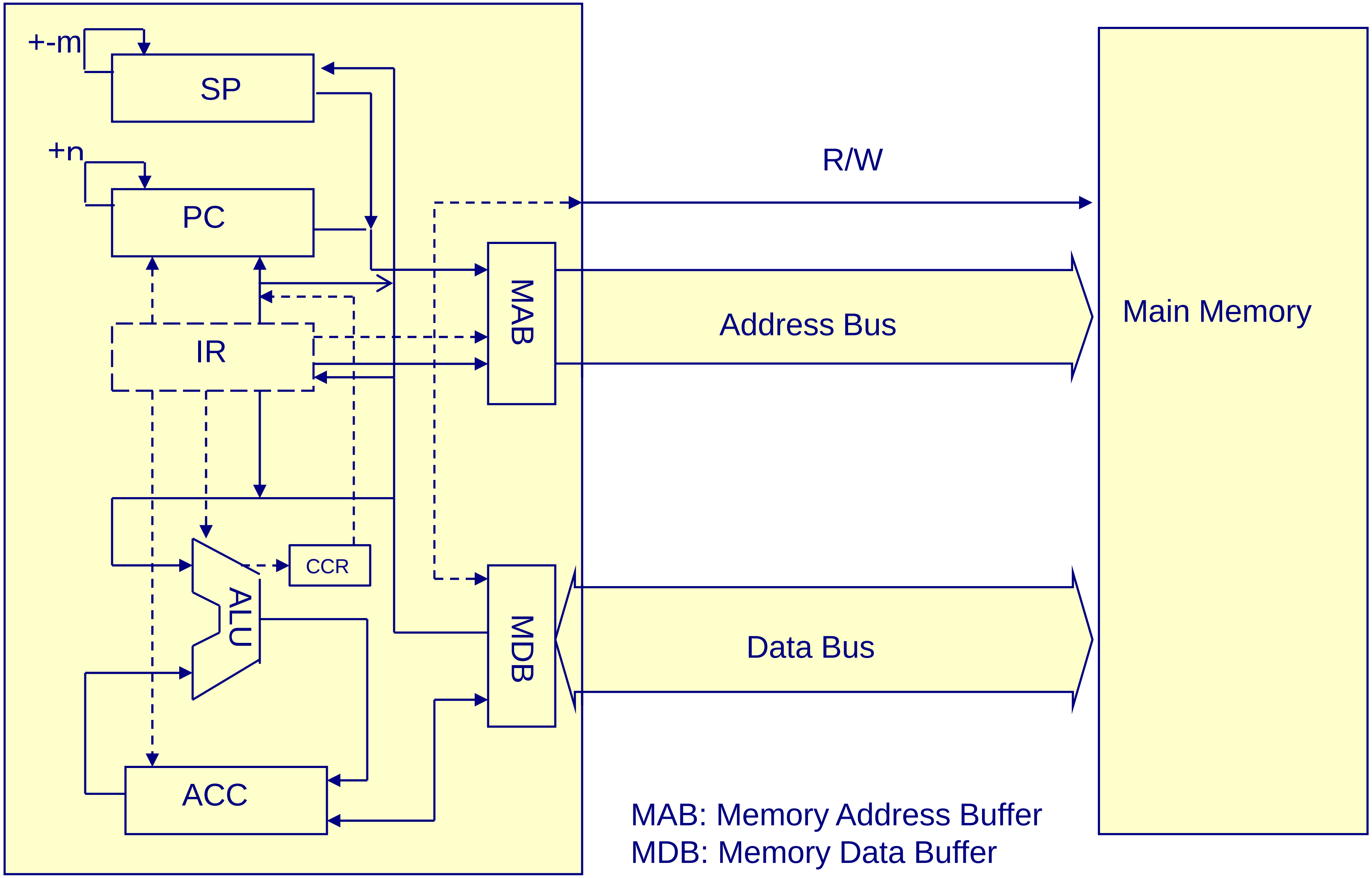
- von Neumann machine consists of
  - addressable memory (MEM) that stores both instructions and data
    - Read: given an address, MEM returns the contents of the storage at the address
    - Write: given an address and contents, MEM writes the contents in the storage at the address
      - An address is assigned to each byte (8 bits)
  - CPU (Central Processing Unit)
    - PC (Program Counter) : stores the address of the instruction (address of the storage that stores the instruction) to be executed next
    - IR (Instruction Register) : stores the instruction currently in execution
      - IR is invisible to programmers
    - ACC (Accumulator) or Registers：stores the data that is an object of an arithmetic/logic operation (called an operand) and the result of the operation
    - Arithmetic Logic Unit (ALU)：applies an arithmetic/logic operation on the operand(s)
    - In addition, there are Stack Pointer (SP) and Condition Code Register (CCR), etc.

# Accumulator Machine

- Accumulator Machine : most CISC (Complex Instruction Set Computer)
- An accumulator machine has a single register, called an accumulator, whose contents are combined with a single operand, with the result of the operation replacing the contents of the accumulator
  - modern machines usually have more than one accumulator: (e.g., Pentium has 4 accumulators)

    accumulator = accumulator OP operand
- To add two numbers in memory and store the result back into memory
  1. place the first number into the accumulator (load operation)
  2. execute the add instruction with the second number as the operand
  3. store the contents of the accumulator back into the memory (store operation)
- How to specify an operand: memory addressing modes
  - Here, we assume that operands for all instructions are accessed by their memory locations (addresses)

# Accumulator Machine (ACC)




MAB: Memory Address Buffer
MDB: Memory Data Buffer

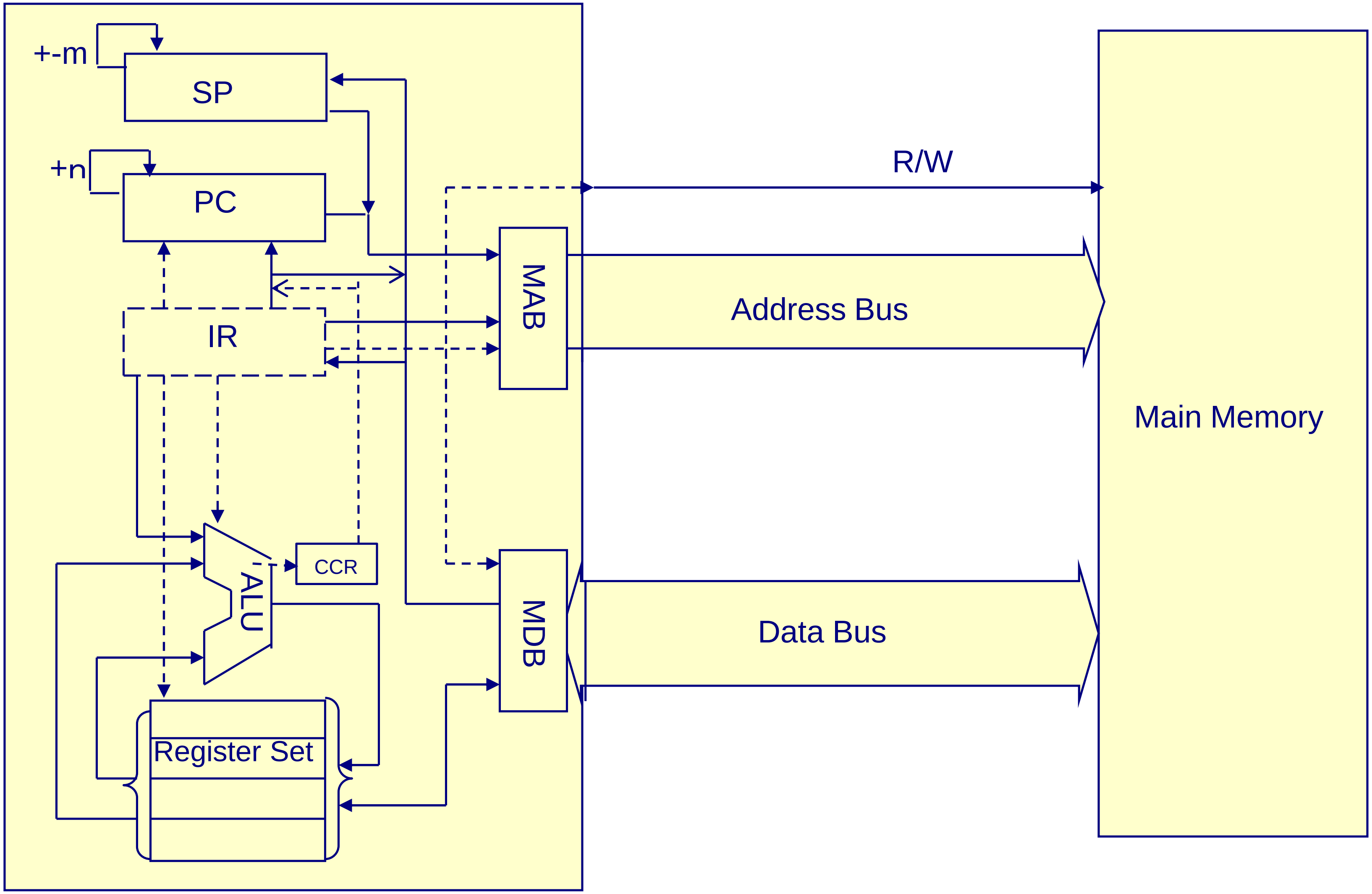
# Register Machine (Load/Store Machine)

- Register machine : most RISC (Reduced Instruction Set Computers)
- CPU has many registers
  — registers provide faster access but are more expensive (compared with memory)
  — they are not as flexible as accumulators
- A small amount of high speed memory, often called a register file, is provided for frequently accessed variables
  — SPARC: 32 registers at any one time
  — H-8: 16 of 8-bit registers or 8 of 16 bit registers
  — This is based on “theory of locality”: at a given time, a program typically accesses a small number of variables much more frequently than others

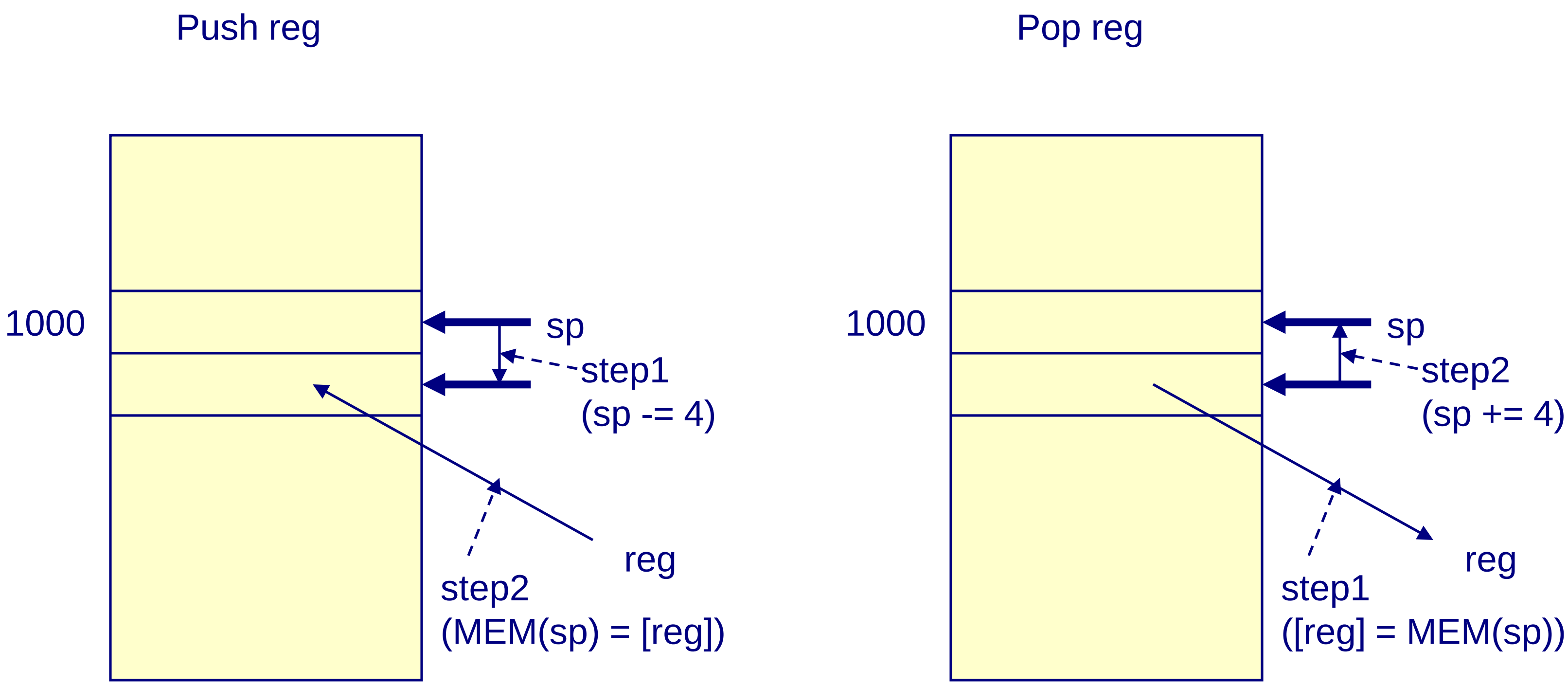
# Register Machine (Load/Store Machine)

- The arithmetic and logic instructions operate with only registers, not with memory.
  - CPU loads and stores the registers from/to memory
- To add two numbers in memory and store the result into memory
  - load the first number into a register, say Reg0,
  - load the second number into another register, say Reg1
  - execute the add instruction; the result is stored in yet another register, say Reg2
  - store the contents of R2 into memory

# Register Machine (Load/Store)

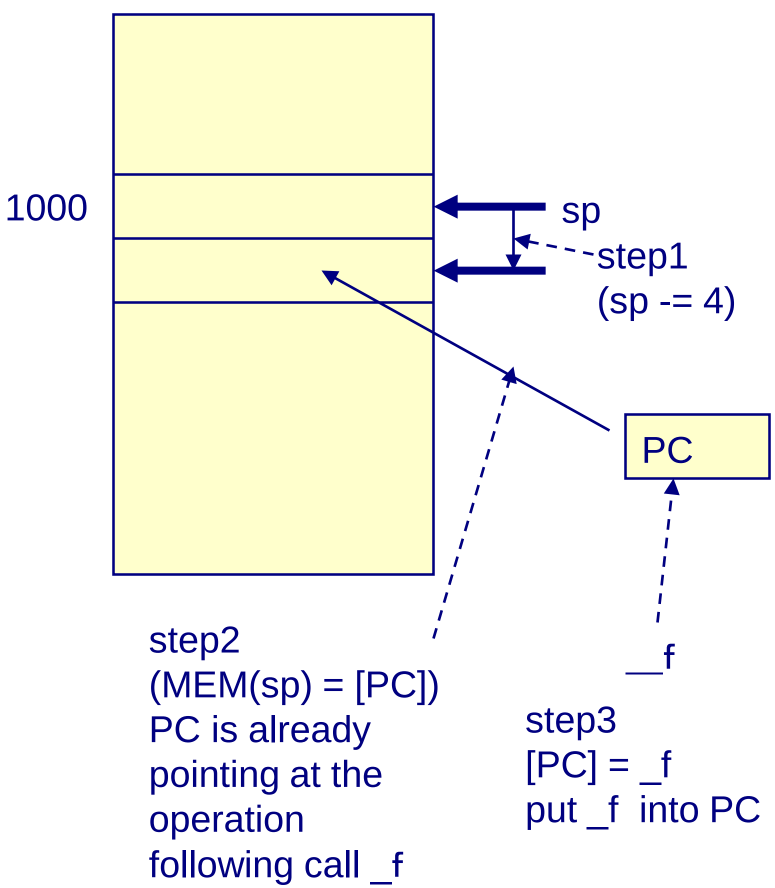


# Stack Operations

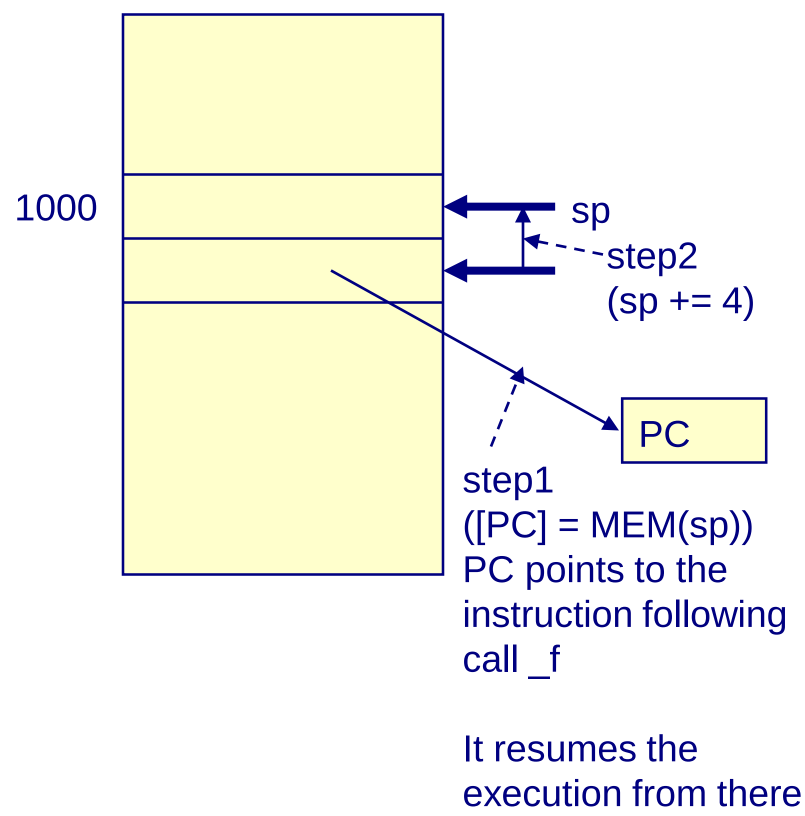


# Call/Ret Operations

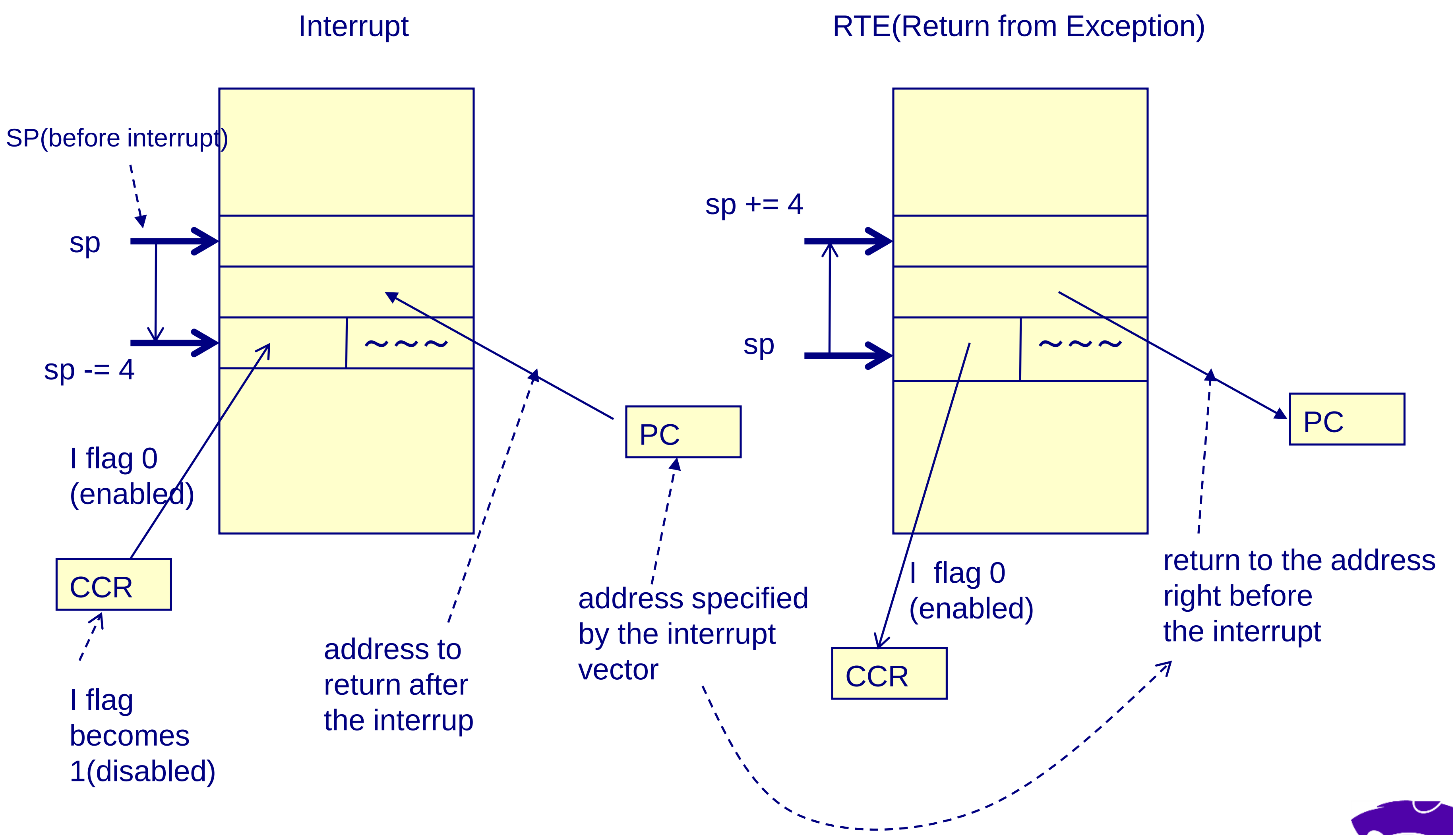
call _f




ret



# Interrupt（H-8）



# Translators and Machine Instructions

- Machine Instructions are classified into the following three categories
  - data transfer operations (memory⟺ register, register ⟺ register, etc)
    - the addressing modes are important
  - arithmetic and logic operations (add, sub, mul, div, and, or, xor, shift, etc)
  - program control operations (jump, call, interrupt, etc)
- Addressing Mode:
  - Resister addressing
  - Immediate addressing →constant
  - Absolute addressing → global and static variables
  - Register indirect addressing → pointers
  - Register relative addressing → automatic (local) variables and parameters

# Advanced C Techniques for Embedded Systems Programming

Basic Computer Architecture

- Execution Environment of C Programs

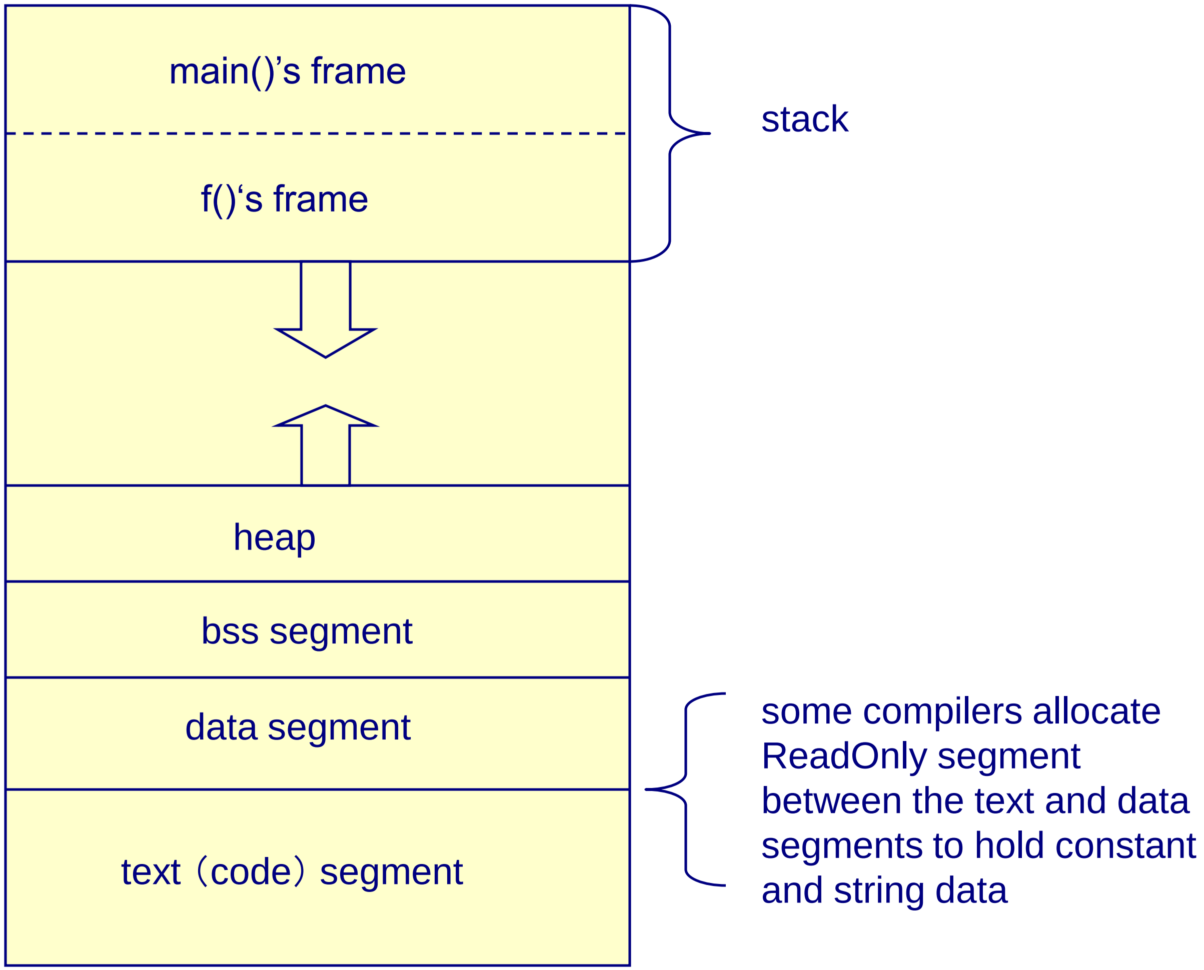
# Storage Classes of C

- External： it is defined outside any function and its scope is basically the whole program
  - from the point at which the variable is defined to the end of the file
  - from the point at which the variable is declared by “extern” to the end of the file
- Static
  - Internal Static： it is defined within a function and its scope is the function
  - External Static：it is defined outside any function and its scope is from the point where it is defined to the end of the file
- Automatic(also called “Local”)： it is defined within a function and its scope is the function
- Parameter (also called “argument”): its scope is the function
- Register： it is a request to the compiler to place the variable in a register (the compiler is free to ignore the request)

# Storage Classes and Memory Allocation

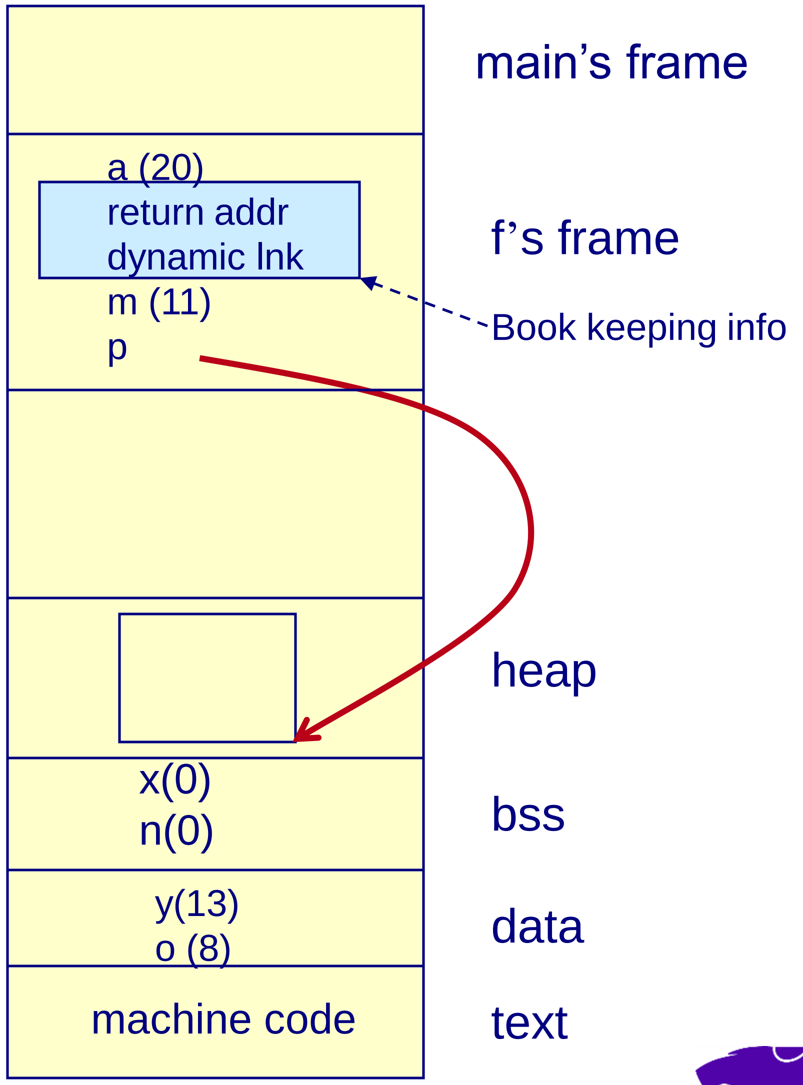
- The execution time memory image of a C program consists of the following five segments (areas):
  - Text segment : stores the machine code of the program
  - Data segment : stores *initialized* “external,” ”external static,” and “internal static” variables
  - BSS segment : stores *uninitialized* “external,” ”external static,” and “internal static” variables (these variables are initialized to 0 before the execution begins)
  - Heap : holds memory areas dynamically allocated by malloc or new
  - Stack : holds function frames
    - when a function is invoked, its frame is allocated (pushed) on the stack
    - when the function returns, the corresponding frame (must be at the top on the stack) is deallocated (popped)
    - a frame stores the function’s actual parameters, automatic variables, and book keeping information (return address, etc)

# The Execution Time Memory Image of a C Program



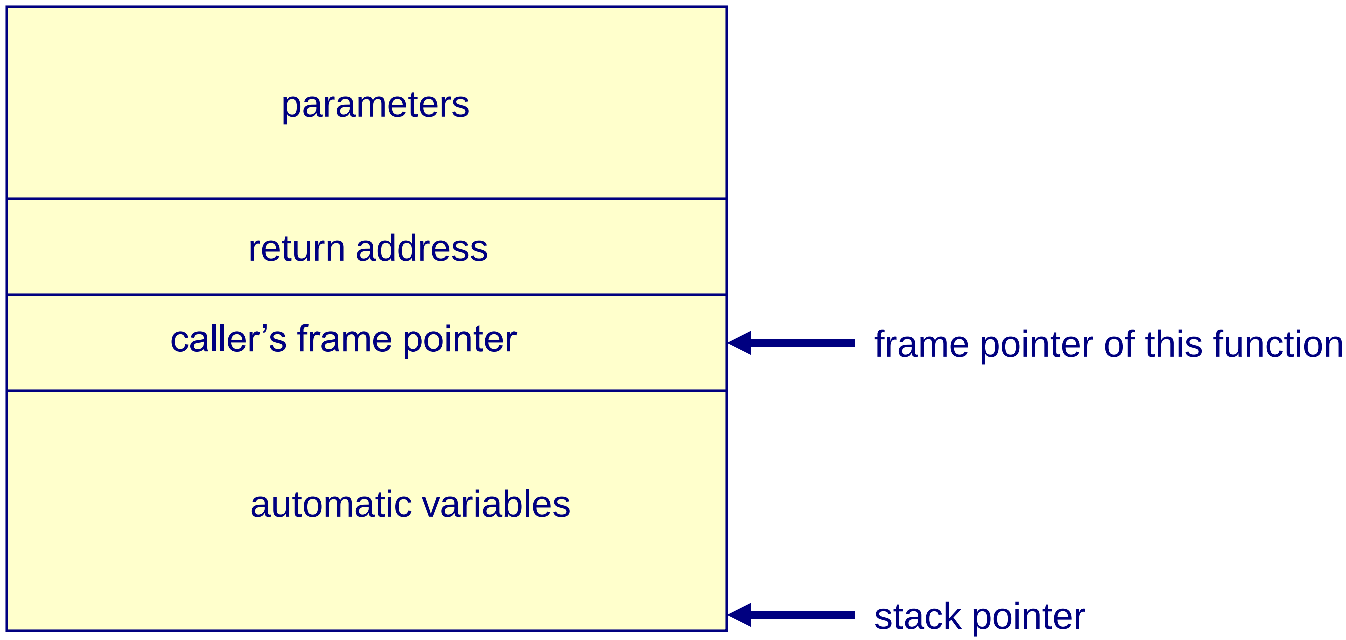
# The Execution Time Memory Image of C Programs (simplified)

```
int x;                    // external
static int y = 13;        // external static

int main() {
  f(20);
  return;
}

void f(int a) {           // parameter
  int m = 11;             // automatic
  static int n;           // internal static
  static int o = 8;       // internal static
  int *p = (int *)malloc(12);  // heap
    ….
}
```



# Frame Structure (Template) of CISC Machines

- CISC （Complex Instruction Set Computer： accumulator-based architecture） such as Pentium and CPU32 have the following frame structure




- The parameters and automatic variables are accessed by relative addressing (frame pointer and offset values)

# Integral Promotion

- Why does the following function call works in C ?

```
char c;
int i;
printf("%d %d\n", c, i);
```

  — this is because a rule called "Integral Promotion" is applied to expressions
    - – if any variable shorter than int (or unsigned int) appears in an expression, the variable is first converted to int (or unsigned int)

- Execute the following program and see the output
  — make sure to understand why the program produces such output

```
void main(){
   char c;
   unsigned char uc;
   unsigned short us1, us2;
   short s1, s2;

   c = 0xf0; uc = 0xf0;
   us1 = c; us2 = uc;
   printf("us1 = %x, \t us2 = %x\n", us1, us2);
   s1 = c; s2 = uc;
   printf("us1 = %x,\t us2 = %x\n", s1, s2);
}
```

# sign extension and zero extension

- For simplicity, we assume to “Integral Promote” a 4 bit data to an 8 bit data
- Integer variables in C have two types: signed (default) and unsigned
  - signed: its MSB (Most Significant Bit （sometimes interpreted as Byte)) is called a “sign bit”: if it is 0, the value is positive; if it is 1, the value is negative
    - Conversion from positive to negative and from negative to positive is done by the following algorithm:
      1. Flip each bit, 1 to 0 and 0 to 1
      2. Add 1 to the LSB(Least Significant Bit)
      - Example 1：how is -6 represented in binary ?
        - +6 is “0110”
        - 1001 (flip each bit)
        - 1010 (add 1 to LSB) this is -6 in binary
      - Example 2：what is “1101” in decimal ?
        - 0010 (flip each bit)
        - 0011 (add 1 to LSB) this is the absolute value of “1101”; thus “1101” is -3

# sign extensionとzero extension (cont)

- unsigned：it always represents positive numbers
  - all 0 is the smallest value and all 1 is the largest value
  - with 4bits, 0000 (0 in decimal) is the smallest value and 1111 (15 in decimal) is the largest value

- The range of the values represented by 4 bits:

| signed | unsigned |
|---|---|
| 0111 (+7) | 1111 (15) |
| 0110 (+6) | 1110 (14) |
| 0101 (+5) | 1101 (13) |
| 0100 (+5) | 1100 (12) |
| 0011 (+3) | 1011 (11) |
| 0010 (+2) | 1010 (10) |
| 0001 (+1) | 1001 (9) |
| 0000 (0) | 1000 (8) |
| 1111 (-1) | 0111 (7) |
| 1110 (-2) | 0110 (6) |
| 1101 (-3) | 0101 (5) |
| 1100 (-4) | 0100 (4) |
| 1011 (-5) | 0011 (3) |
| 1010 (-6) | 0010 (2) |
| 1001 (-7) | 0001 (1) |
| 1000 (-8) | 0000 (0) |

# sign extensionとzero extension (cont)

- Consider how to Integral Promote a 4 bit data to an 8 bit data
- The important things are (1) to maintain the same signed/unsigned property and (2) to maintain the same value, after applying Integral Promotion
- First, consider an unsigned variable
  - It is easy to see that the value is maintained if 0 is put into the top 4 bits
    - Example：1101 → 00001101 (both are 13）
- Now, consider a signed variable
  - if the value is positive, it is easy to see that the value is maintained if 0 is put into the top 4 bits
    - Example：0101 → 00000101 (both are 5）

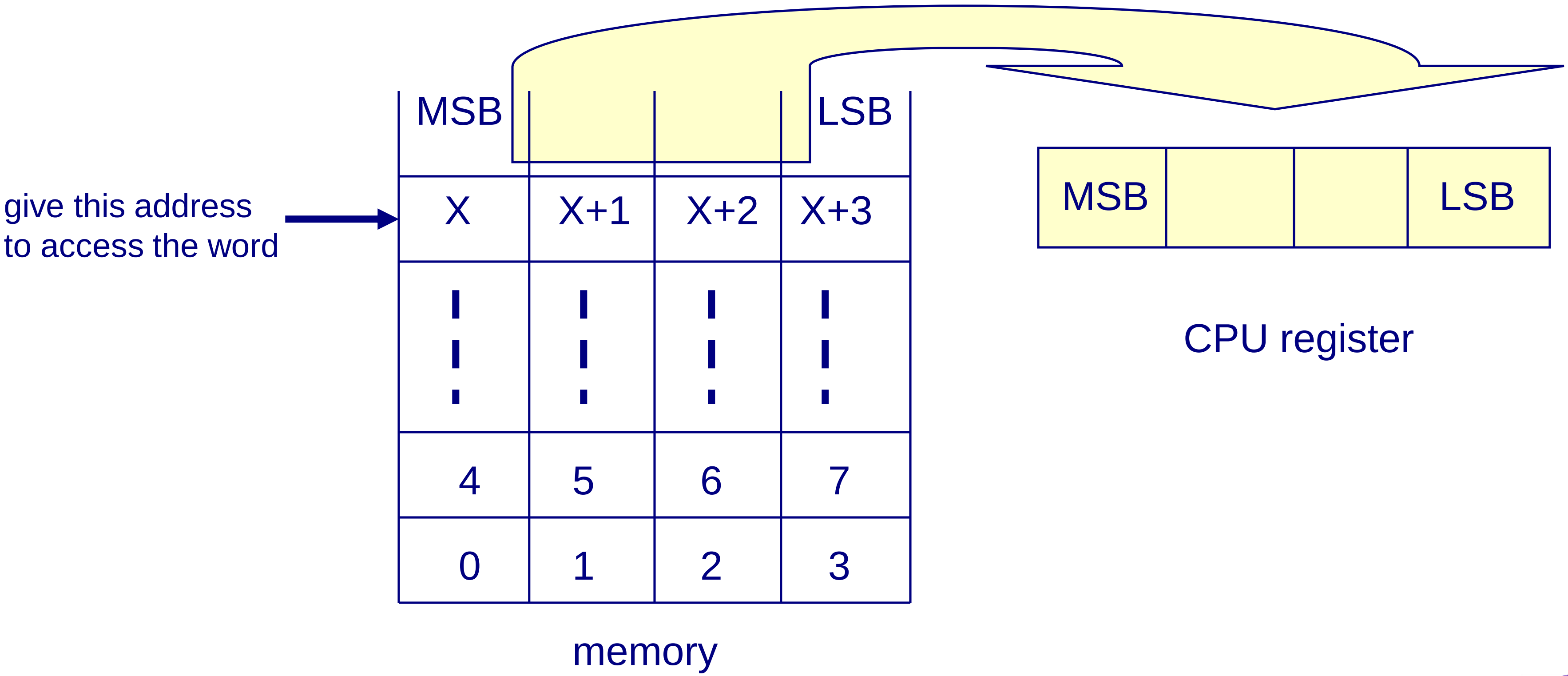
# sign extensionとzero extension (cont)

- How about a negative value ?
  - Example：if 0 is put into the top 4 bits of 1011 (-5), the 8-bit value becomes 00001011, which is positive 11
  - If 1 is put into the top 4 bits instead, the same value is maintained
  - Example：1011 (-5) → 11111011
    - “11111011” is a negative value.
    - Its absolute value is obtained by flipping each bit followed by adding 1 to its LSB (00000101(+5)), and the same value is maintained
  - That is, if the value is positive (more precisely, non-negative), 0 should be put, and if the value is negative, 1 should be put in the top 4 bits.

- Conclusion:
  - For a signed variable, extend its sign bit (MSB) to the top bits
    - This is called ”sign extension”
  - For an unsigned variable, always put 0 to the top bits
    - This is called ”zero extension”

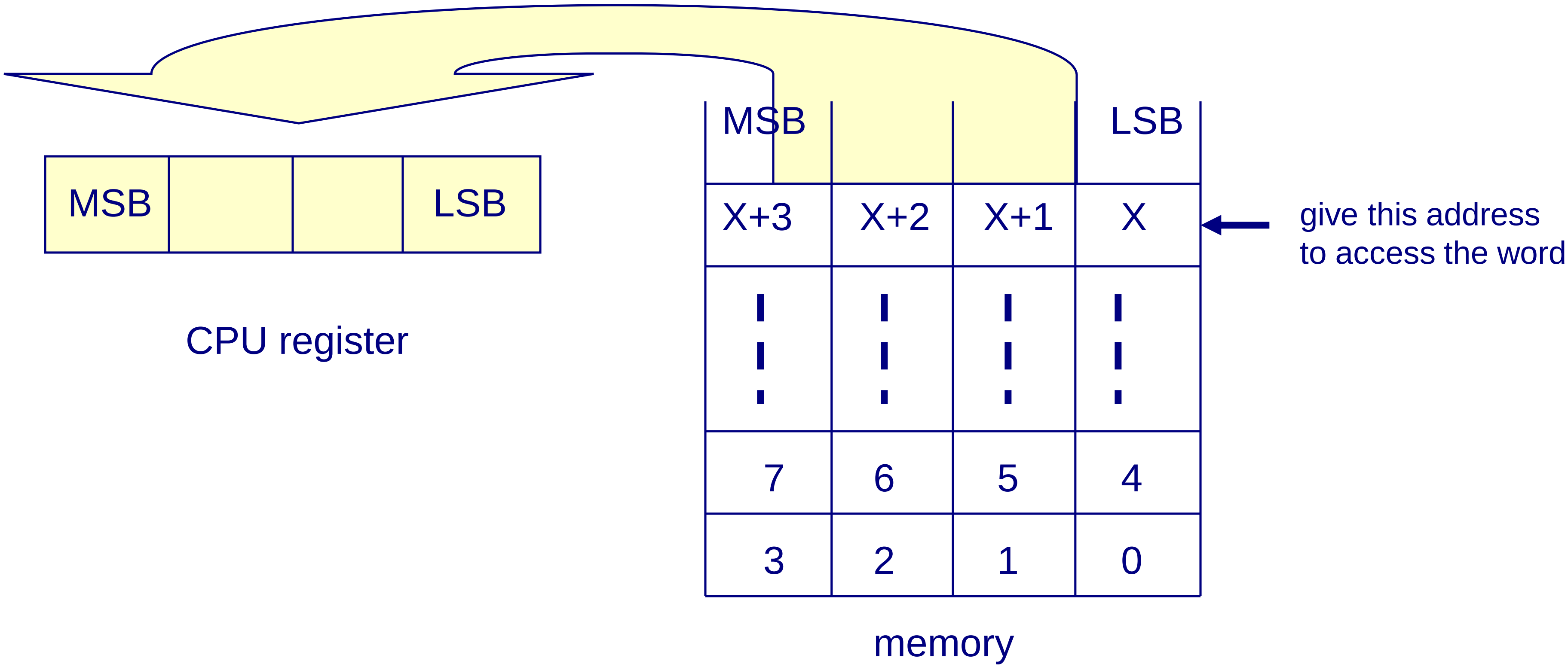
# Big Endian and Litte Endian

- There are two types of allocation of bytes in a word, called Big Endian and Little Endian
    1. Big Endian (CPU32、SPARC)



# Big Endian and Litte Endian (cont)

2. Little Endian (Pentium、C167)



# Testing Endianness using pointers

- Endianness can be checked by executing the following program:

```
int x = 0x12345678;

void main() {
   char *cp = (char *) &x;
   printf("x\t%x\t%x\t%x\n", *cp, *(cp+1), *(cp+2), *(cp+3));
}
```

- Execution results on Pentium

  78 56 34 12

- Execution results on Sparc

  12 34 56 78

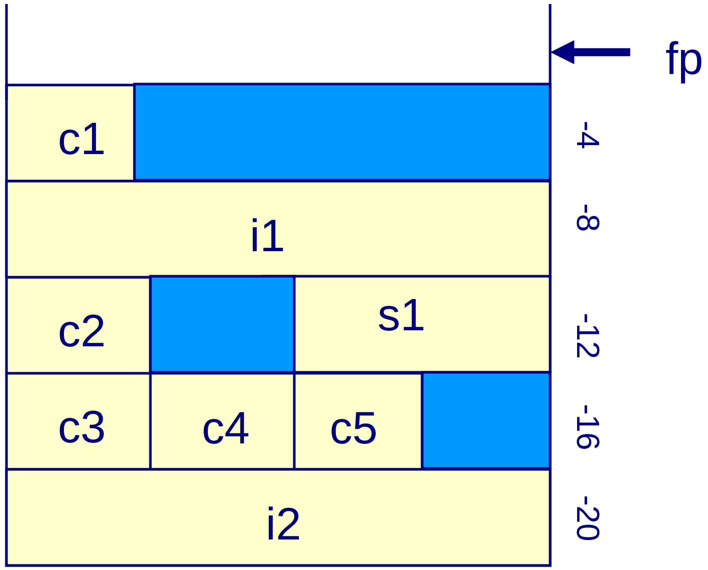
- Test the endianness of H-8

# Allocation of Variables

- 2-byte variables (short and int on16bit machines) are accessed quicker if they are allocated at a 2-byte boundary (even address)
  - compilers allocated those variables at a 2-byte address boundary
- Similarly, 4-byte variables (long, float, and int on 32bit machines, etc) are accessed quicker if they are allocated at a 4-byte boundary
  - compilers allocate those variables at a 4-byte address boundary
  - if not, address errors will result on some machines (e.g., SPARC)

Consider the following declarations:

```
char c1;
int i1;
char c2;
short s1;
char c3;
char c4;
char c5;
int i2;
```




Note: this is not always the case

# Function Call on a CISC Machine

Caller

1. push parameters in the reverse order on Stack
2. “call (jump subroutine to) function” – one machine instruction which does the following:
   – push the return address (the address following this instruction (3) on Stack)
   – go to the function — set the starting address of the function in Program Counter
3. remove the parameters from Stack (esp += sizeof(parameters))
4. use the return value found in the major register(s) (e.g., eax or (eax, edx) pair in Pentium)

# Function Call on a CISC Machine (cont)

Callee

A. push Caller’s Frame Pointer(ebp) on Stack
B. update Frame Pointer to point to this frame (ebp = esp)
C. allocate automatic variables (esp-=sizeof(automatic variables))

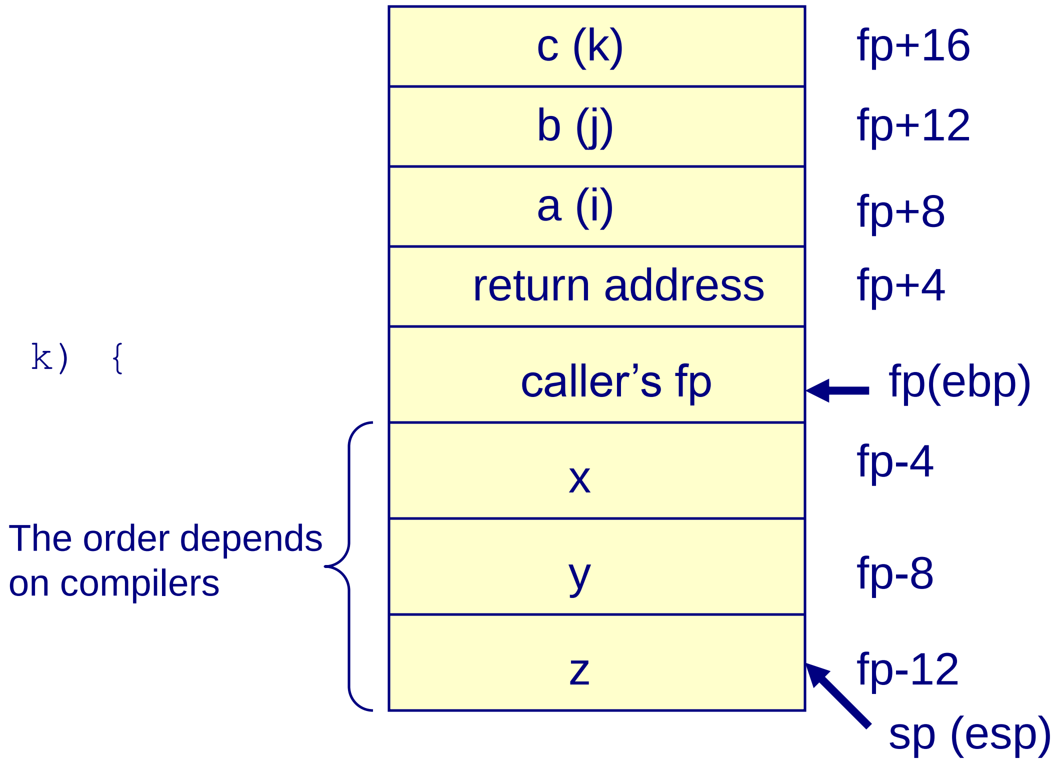
body of the function

D. set the return value in the major register(s)
E. de-allocate automatic variables (esp = ebp)
F. restore Caller’s Frame Pointer ( pop ebp)
G. “return from function” – one machine instruction which does the following
   - pop the return address (3) found on Stack and jump there (set the address in Program Counter)

# Allocation and De-allocation of a Frame (cont)

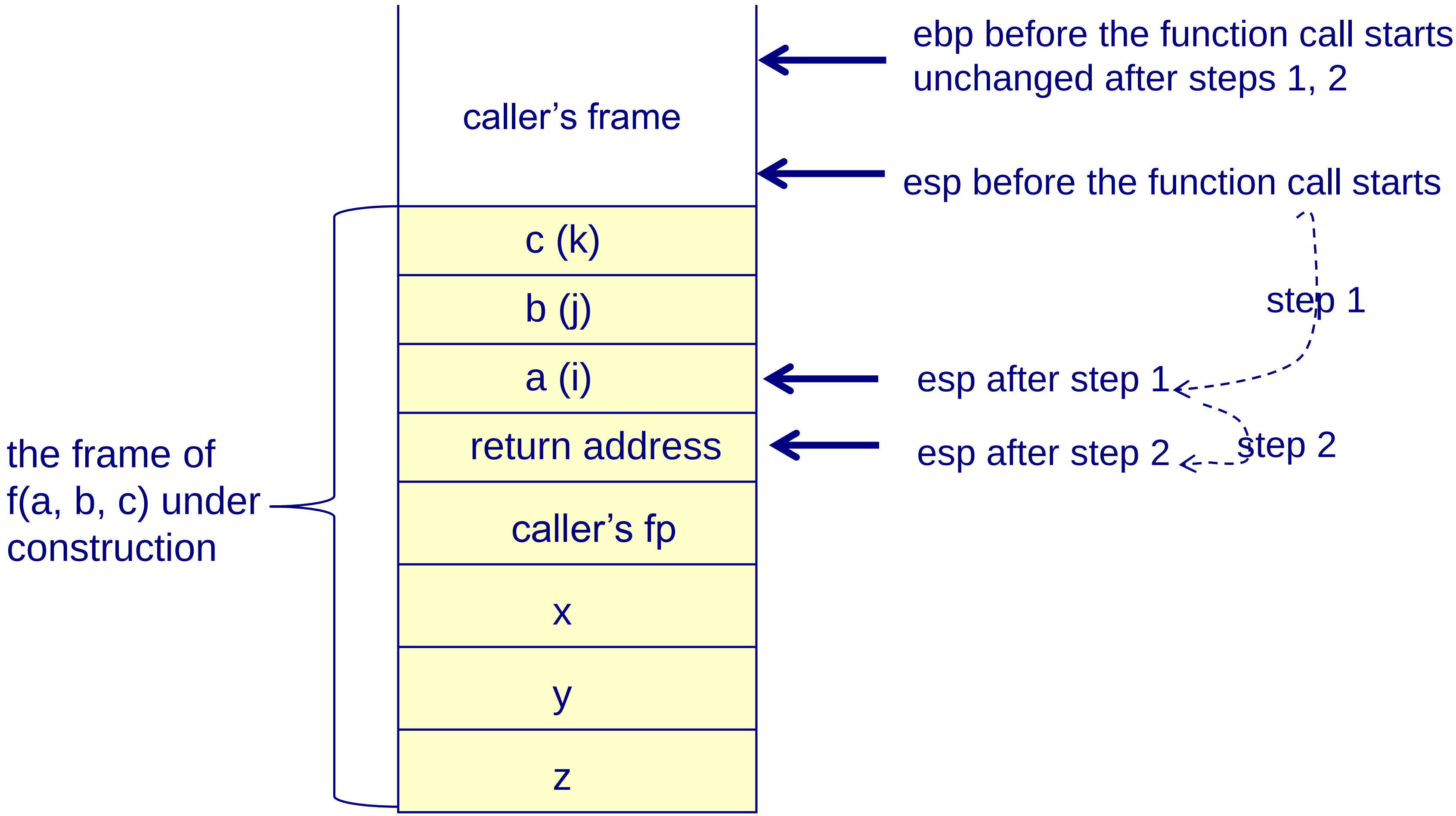
- Consider the execution of the following program (Pentium)

```
int m = f(a, b, c);



int f(int i, int j, int k) {
    int x, y, z;
            .
            .
            .
    return x + j;
}
```

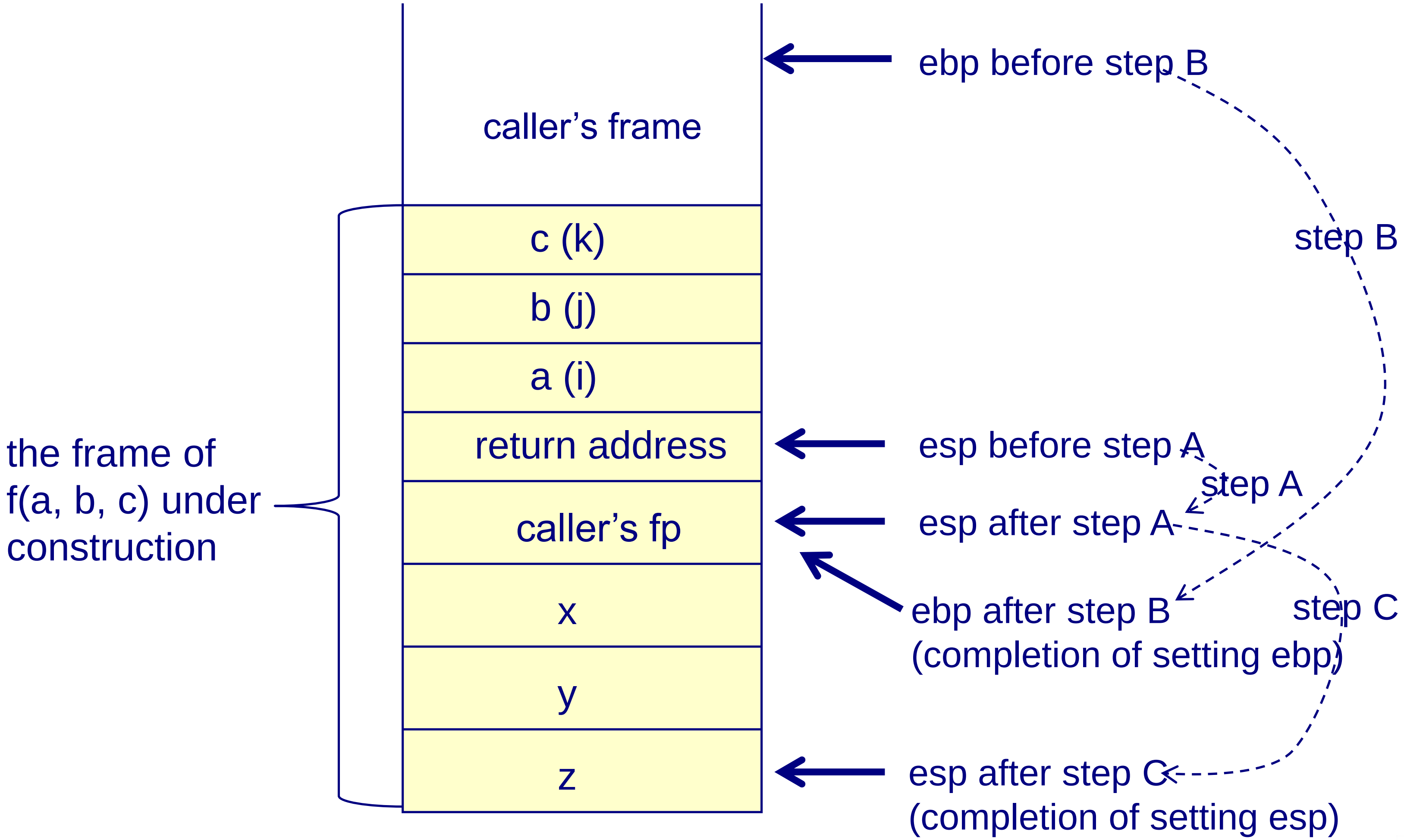



f(a, b, c)'s frame

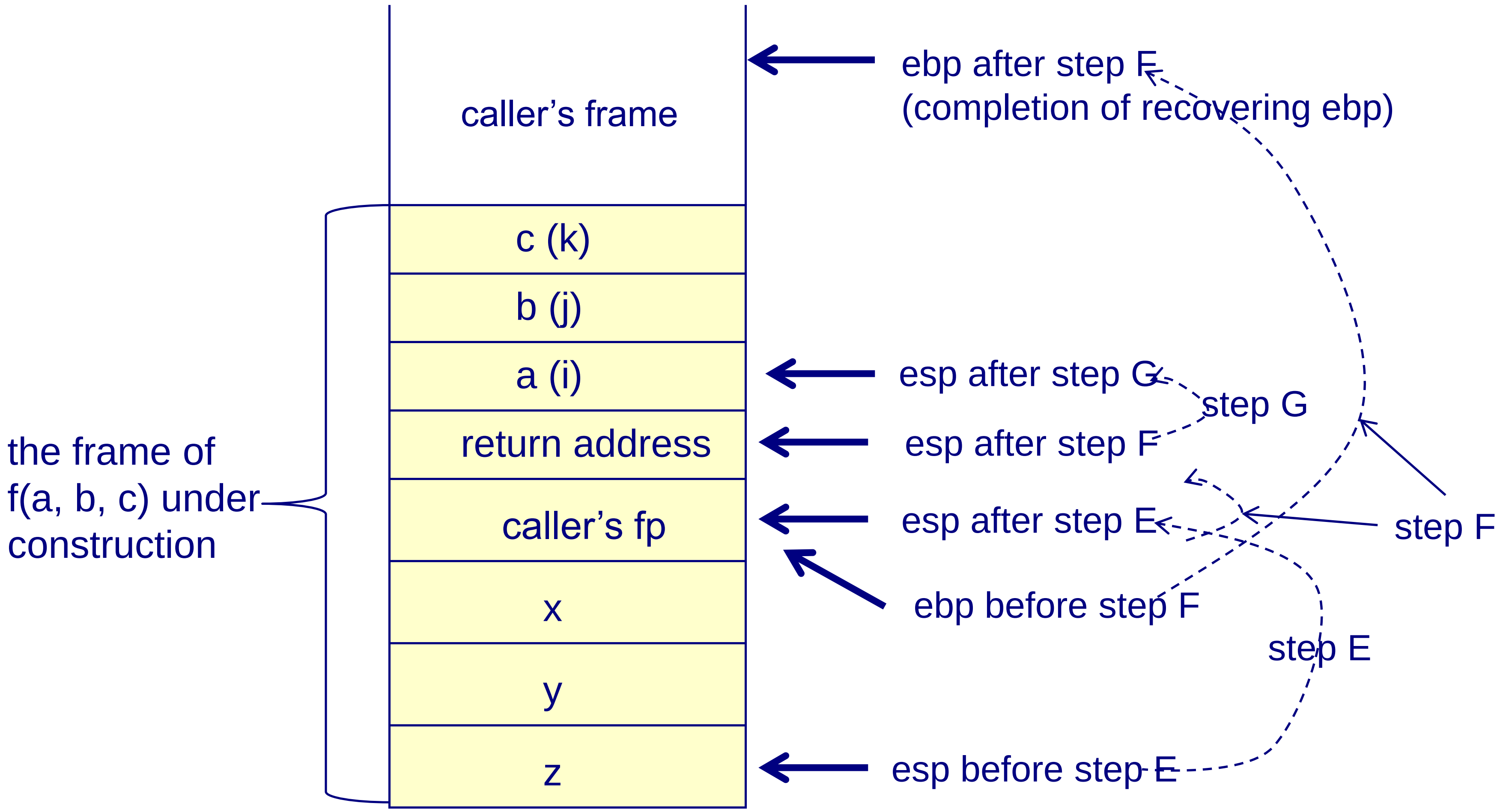
# Frame Allocation Sequence (by caller)



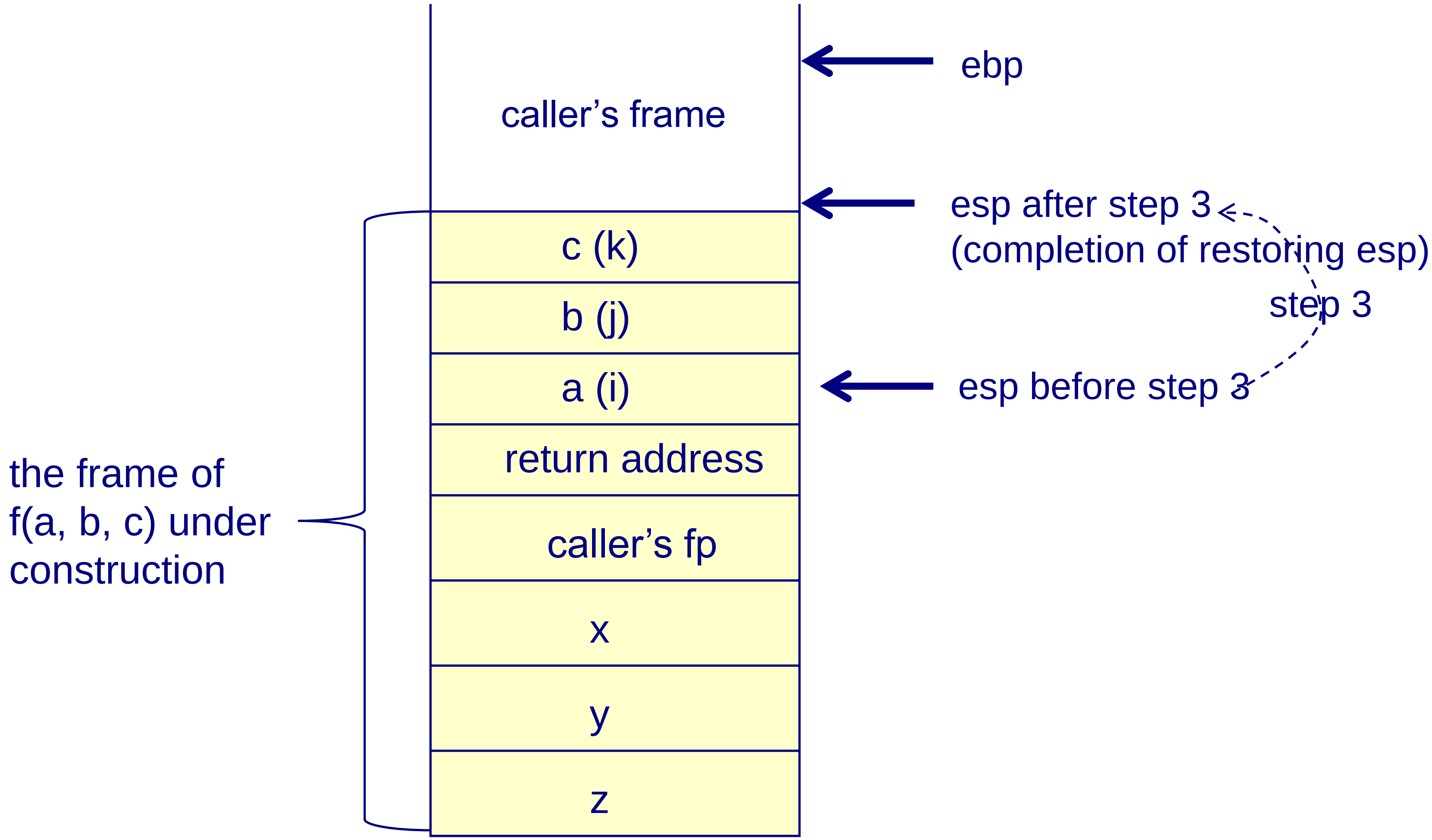
# Frame Allocation Sequence (by callee)



# Frame Deallocation Sequence (by callee)



# Frame Deallocation Sequence (by caller)

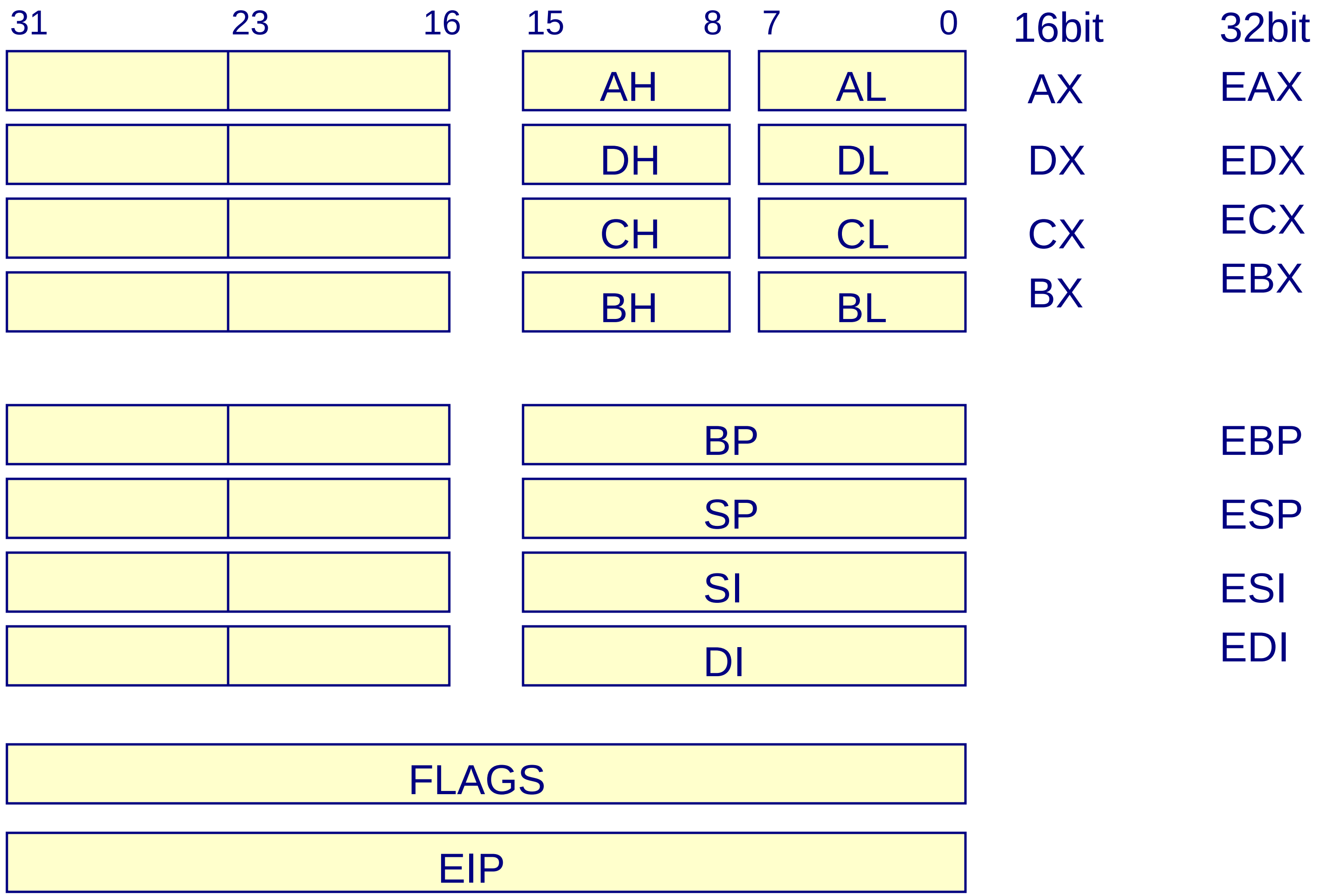


# Advanced C Techniques for Embedded Systems Programming

- Basic Computer Architecture
- Execution Environment of C Programs
  - on Pentium
  - on Hitachi H8
- Pointer Arithmetic and Arrays

# Pentium Register Organization

- Segment Registers do not need to be considered for programs running on OS



# Effective Address Calculation of Pentium

- On Pentium, the address is determined by the sum of the following three elements:
  1. Displacement (constant value embedded in the instruction)
  2. Value in Base register
  3. Value in Index register * scaling factor(1,2,4,8)

| Base | + | (Index | * | scale) | + | Displacement |
|---|---|---|---|---|---|---|
| eax | | eax | | 1 | | 8-bit displacement |
| ebx | | ebx | | 2 | | 32-bit displacement |
| ecx | | ecx | | 4 | | |
| edx | | edx | | 8 | | |
| esp | | | | | | |
| ebp | | ebp | | | | |
| esi | | esi | | | | |
| edi | | edi | | | | |

# VC++.NET Assembler

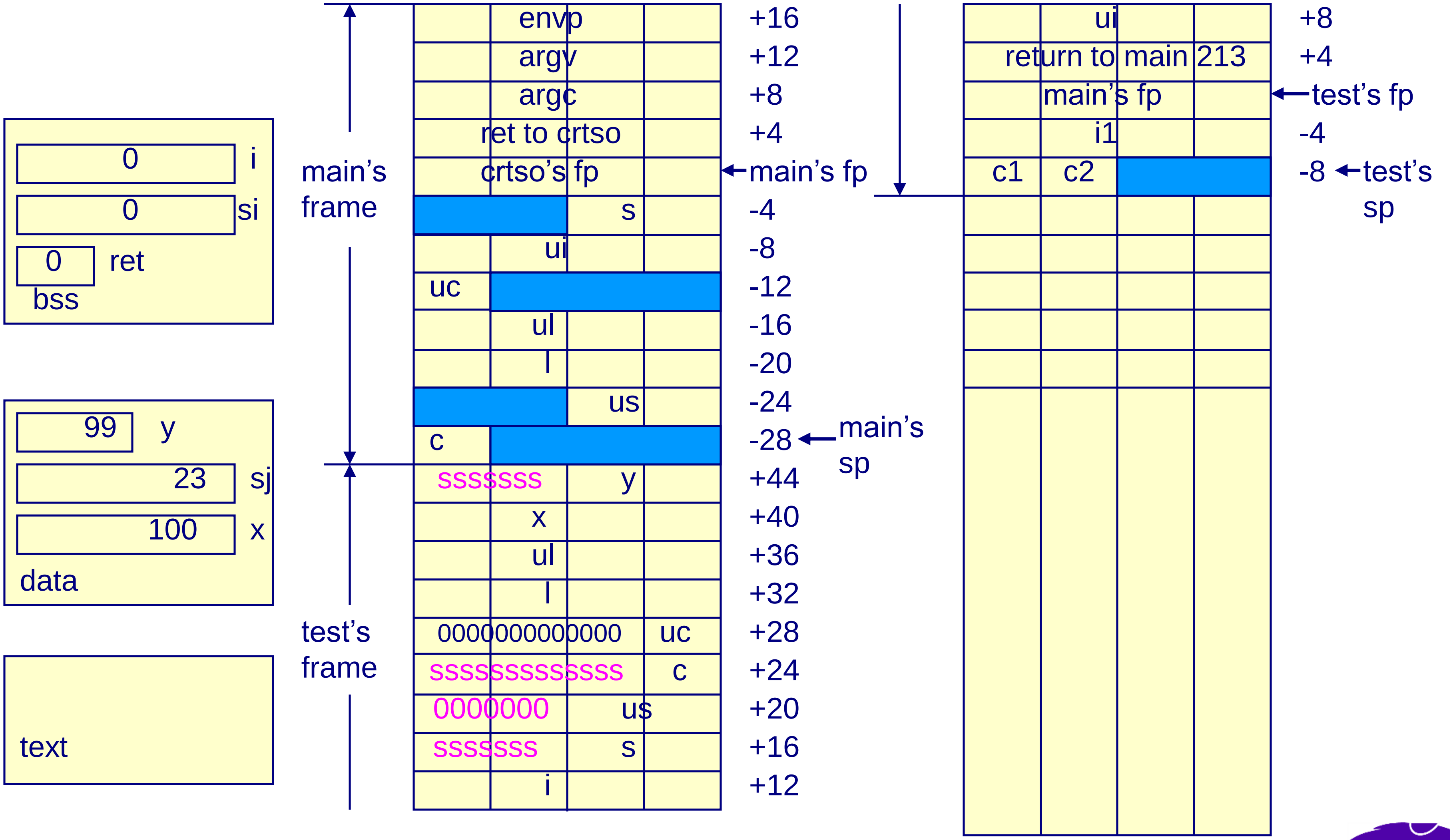
- How to generate assembly code of file.c in VC++ compiler
  - open the VS.NET2017 command prompt
  - cl /Od /FAcs file.c
    - /Od : no optimization
    - /FAcs : generate assembly listing including source and machine code

- Pseudo instructions of the VC++ assembler:
  - DD : Define Double (allocate 4 bytes and initialize the area)
    - _x DD 064H
  - DW : Define Word (allocate 2 bytes and initialize the area)
    - _y DW 063H
  - PUBLIC : export a label (label: a literal associated with an address)
    - PUBLIC _x
  - COMM : COMMUNAL (allocate the specified number of bytes and export the label (note: even though it is defined in the data or text segment, the allocation takes place in the bss segment)
    - COMM _ret:Byte

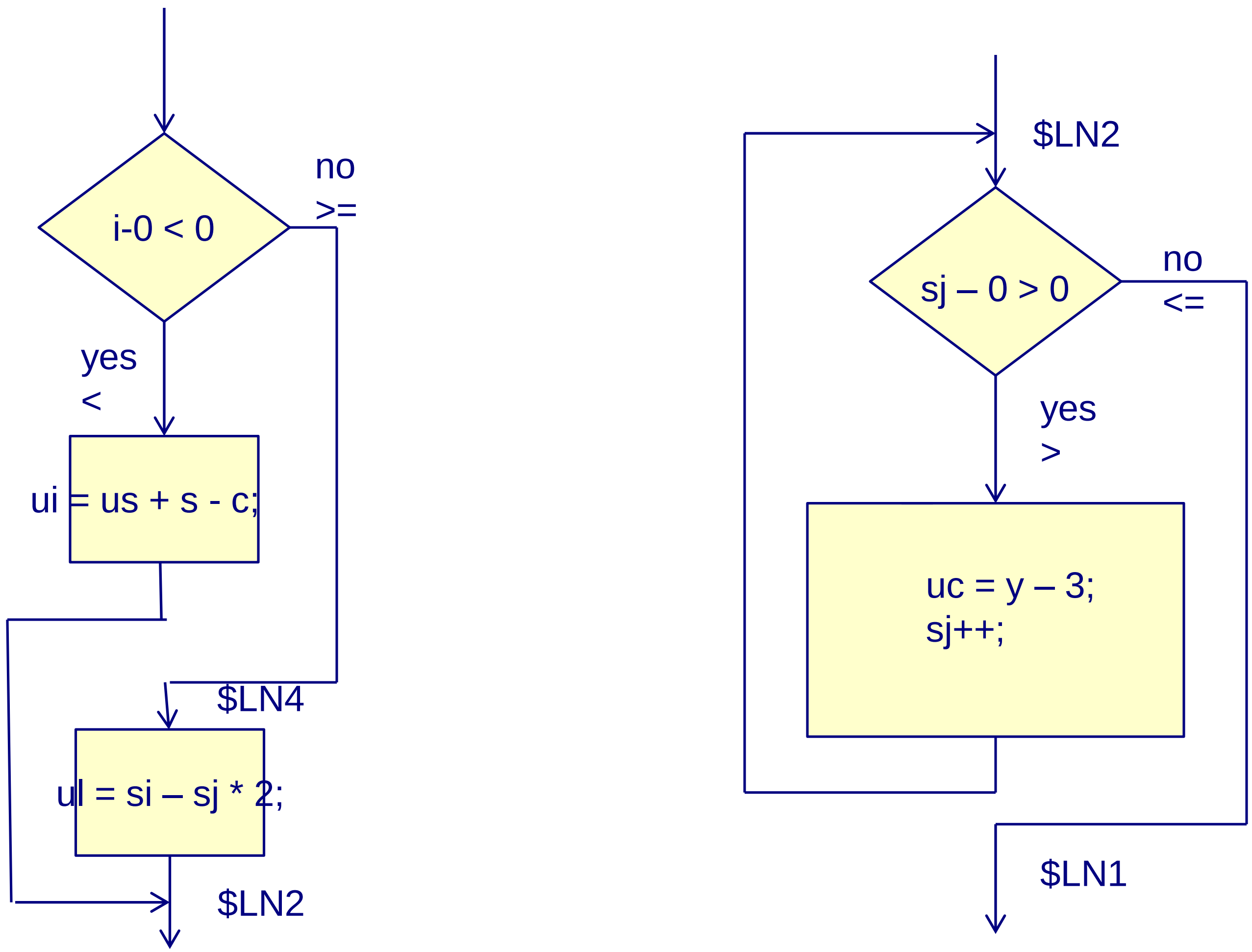
# VC++.NET Assembler (cont)

- operation dest, src
  - in the subtract operation, dest = dest-src
- WORD PTR “effective address”: access a word (2byte data) located at “effective address”
  - mov ecx, WORD PTR _ui$[ebp]
- DWORD PTR “effective address”: access a dword (4byte data) at “effective address”
- BYTE PTR “effective address”: access a byte (1byte data) at “effective address”
- OFFSET “effective address” : denotes the address of an element, not the contents of the element (WORD PTR) or the constant
  - push OFFSET $SG3065
- number DUP(value): multiple initialization of the “number” of elements with “value”
  - DUP(?) : the initial value is not specified (uninitialized)
    - _g2 DD 01H DUP (?)

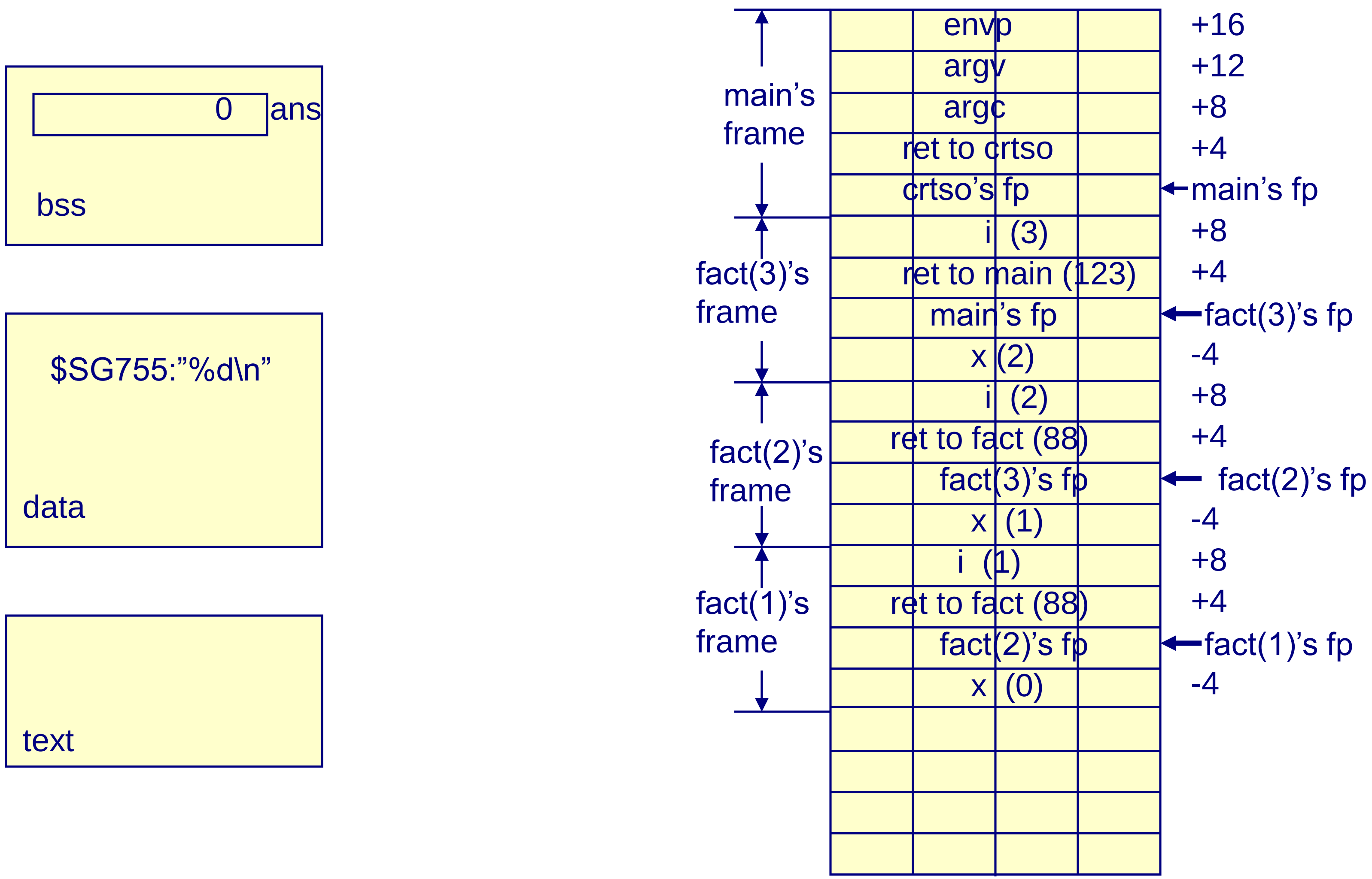
# Exectution of var.c on Pentium



# If and While statements (var.c)



# Execution of fact.c on Pentium



# short cut evaluation (fact.c)