Chapter 1_2

Advanced C Techniques for Embedded Systems Programming

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 (?) // for “static int g2;”
    - _x DD 064H DUP (?) // for “static int x[100];

Variable allocation by VC++

- External variable
  - with initializer (in data)

    PUBLIC _x

    _x DD 064H // for "int x = 100;"
  - without initializer (in bss)

    COMM _ret:BYTE // for "char ret;"

    // COMM exports the name
- Static variable (internal static, external static)
  - with initializer (in data)

    _sj DD 017H // for "static int sj = 23;"
  - without initializer (in bss)

    _si DD 01H DUP (?) // for "static int si;"

    // since COMM exports the name, COMM cannot be used here

  Note: Some assemblers has LCOMM (for local comm) to allocate static variables in bss

Advanced C Techniques for Embedded Systems Programming

- Execution Environment of C Programs on Pentium (x86)
  - VC++
  - Minix cc
  - gcc

Execution Environment of C program on 64 bit Pentium x86_6

Execution Environment of C Programs on H-8

Destructive and non-destructive registers

- Compilers often manage registers by grouping them into the following two classes:
  - Destructive registers
    - Their values may change in an invocation of a function
  - Non-destructive registers
    - Their values are guaranteed to be the same before and after an invocation of any function
    - If a function needs to use a non-destructive register, in general, code generated by a compiler would be
      - save (push onto the stack) the value in the non-destructive register being used in the function
      - function body
      - restore (pop from the stack) the saved value back to the non-destructive register
      - return from function

Destructive and non-destructive registers (cont)

- Example:

R2 ← 4 // destructive register
R4 ← 20 // non-destructive register
call func( )
{R2 == don't know && R4 == 20}

In func( ), if R4 (non-destructive register) is used
push R4
function body
pop R4
return

Destructive and non-destructive registers (cont)

- In most compilers for Pentium, ESI, EDI are non-destructive registers
  - In Pentium gcc compiler, EBX also seems a non-destructive register
  - But in Minix compiler, EBX is not a non-destructive register
- Register variables are assigned only to non-destructive registers

Minix Assembler pseudo instructions, etc.

- operation dest, src
  - in the subtract operation, dest = dest-src
- .extern: export the label name
- .data4 value : allocate 4 bytes and initialize the area with “value”. The name (label) is not exported
  - According to the documentation, minix assembler has .data2 and .data1. But the compiler does not use these pseudo instructions
  - int x = 100; is compiled to

```
.export _x
_x:
.data4 100
```

Minix Assembler pseudo instructions, etc. (cont)

- .comm label, num: allocate “num” bytes and name the area “label”. (Unlike VC++ compiler), the name is not exported
  - static int si; is compiled to

    .comm _si, 4
  - char ret; is compiled to

    .extern _ret

    .comm _ret, 4   // even though 1 one byte should be allocated,

    // allocate 4 bytes compiler

- To access 1-, 2-, and 4-byte variables:

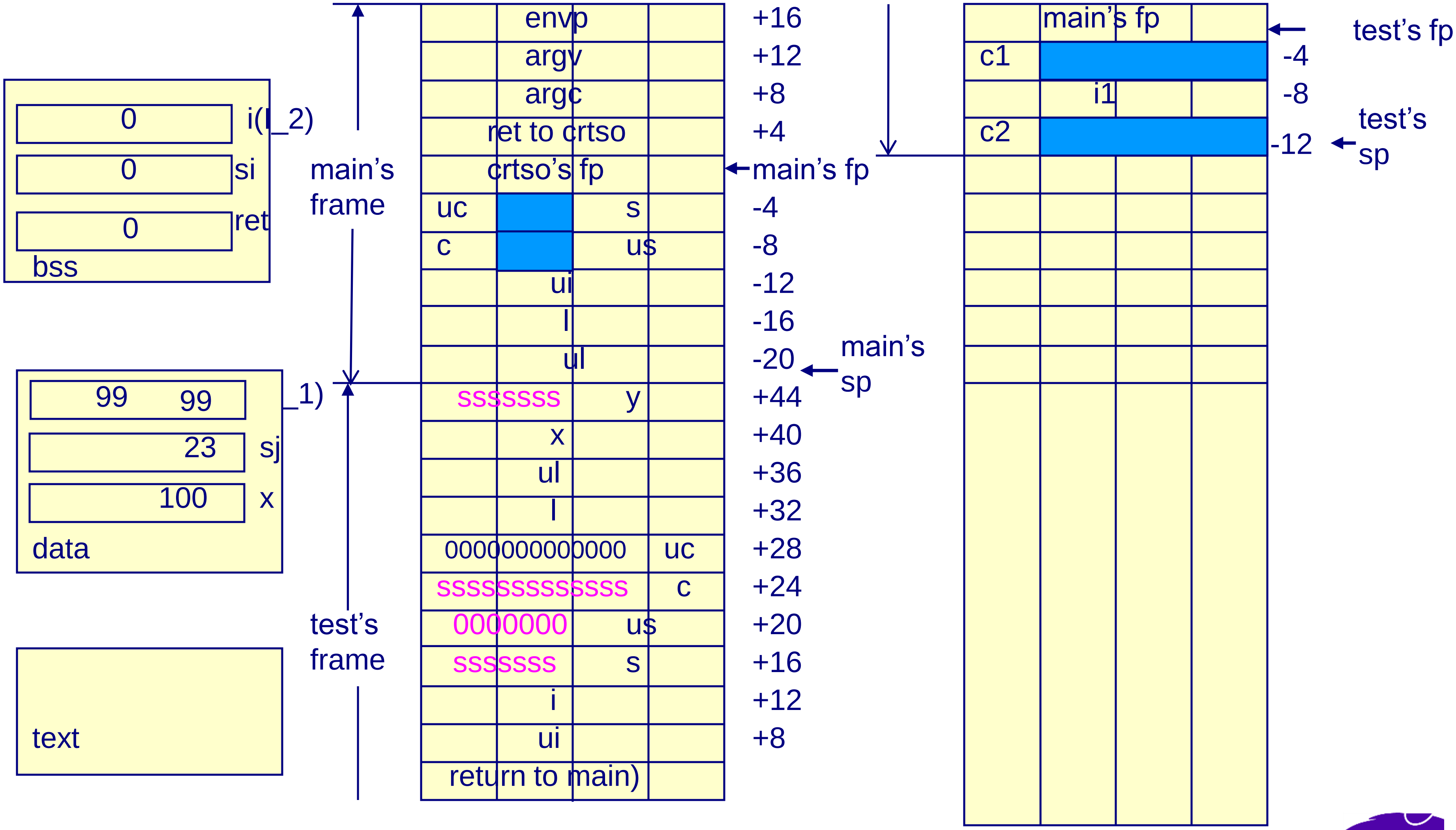
  mov   8(ebp), 1   // move constant 1 to 4-byte area at 8(ebp)

  016 mov   16(ebp), 3   // move constant 3 to 2-byte area at 16(ebp)

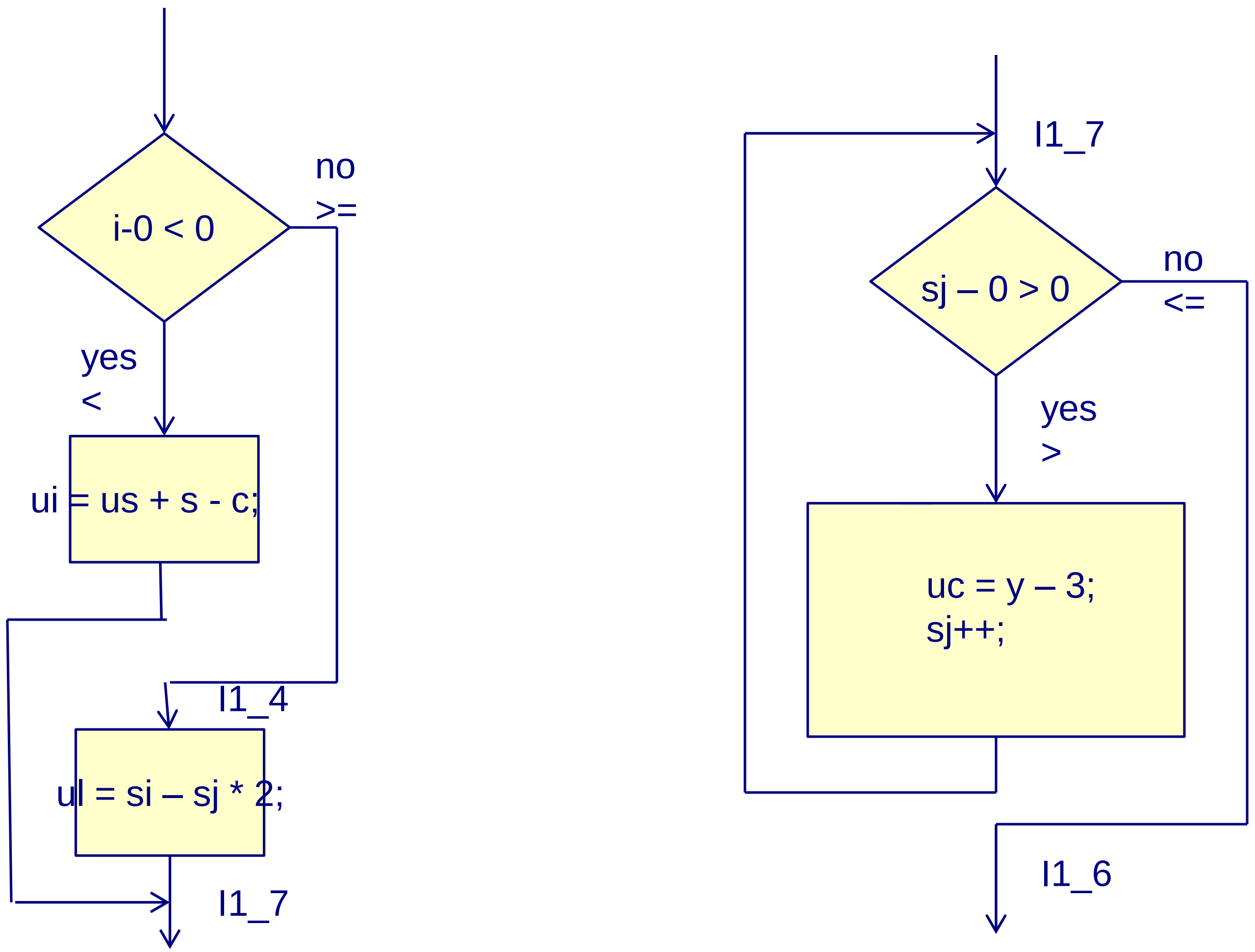
  movb   24(ebp), 5   // move constant 5 to 1-byte area at 24(ebp)

- Unlike gcc for x86, the non-destructive registers are only esi and edi
  - ebx is a destructive register

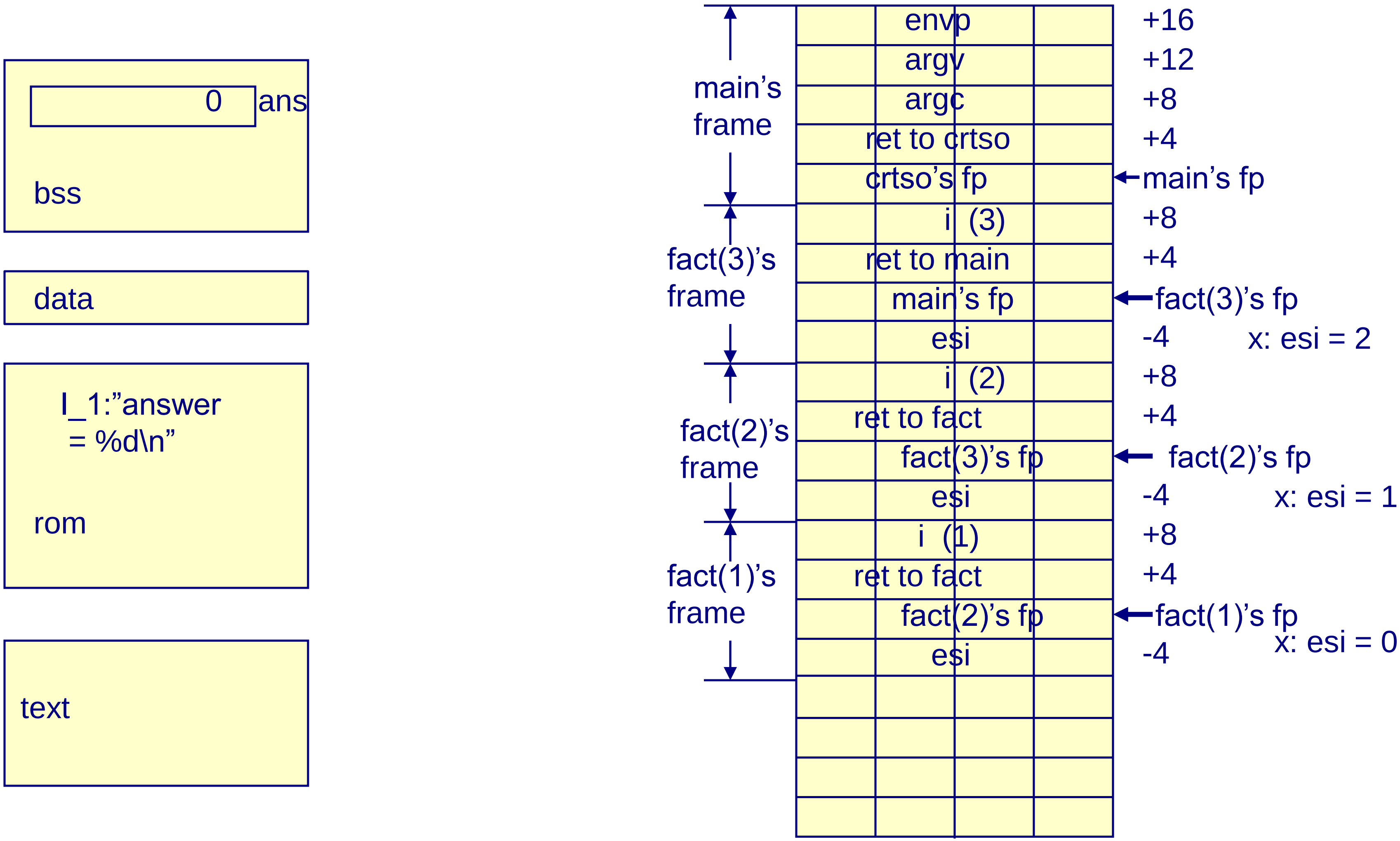
Exectution of var.c on Pentium (Minix cc)



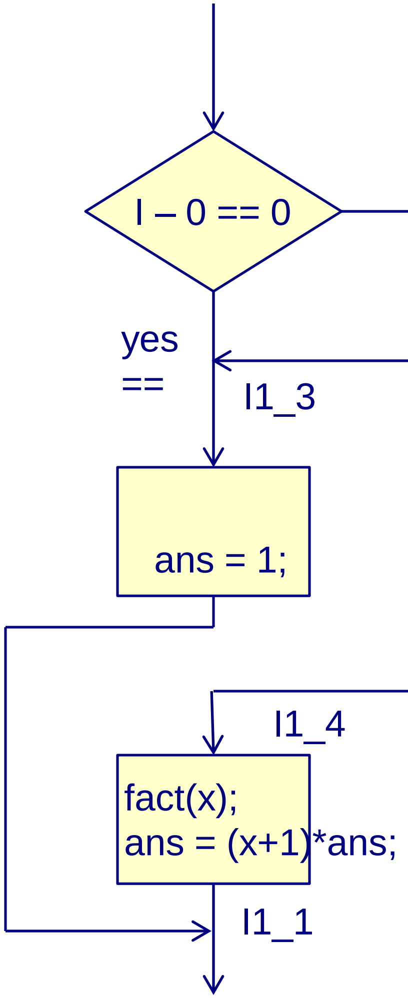
If and While statements (var.c) (Minix cc)



Execution of fact.c on Pentium (Minix cc)



short cut evaluation (fact.c) (Minix cc)



- Execution Environment of C Programs on Pentium (x86)
  - VC++
  - Minix cc
  - gcc

Execution Environment of C program on 64 bit Pentium x86_6

Execution Environment of C Programs on H-8

Intel VS ATT assembly format

- In default, gcc generates assembly code in ATT format, whereas VC++ generates code in Intel format.
  — to generate code in Intel format in gcc, “gcc -O0 –S –masm=intel file.c”

- The differences:
  1. Specification of source and destination operands
     – In Intel: operation dest, source
     – in ATT: operation source, dest
  2. Specification of registers
     – In Intel: eax, edx, ecx, ebp, esp etc
     – In ATT: %eax, %ecx, %dbp, %esp, etc
  3. Specification of length of operands
     – In Intel: mov ecx, DWORD PTR _ui$[ebp] etc
       mov WORD PTR _us$[ebp], 4 4: immediate address (constant)
     – In ATT: movl -8(%ebp), ecx etc
       // l for long (4 bytes), w for word (2 bytes), and b for byte (1 byte)
       movw $4, 24(%ebp) $4: immediate address (constant)

Notes on gcc compiler

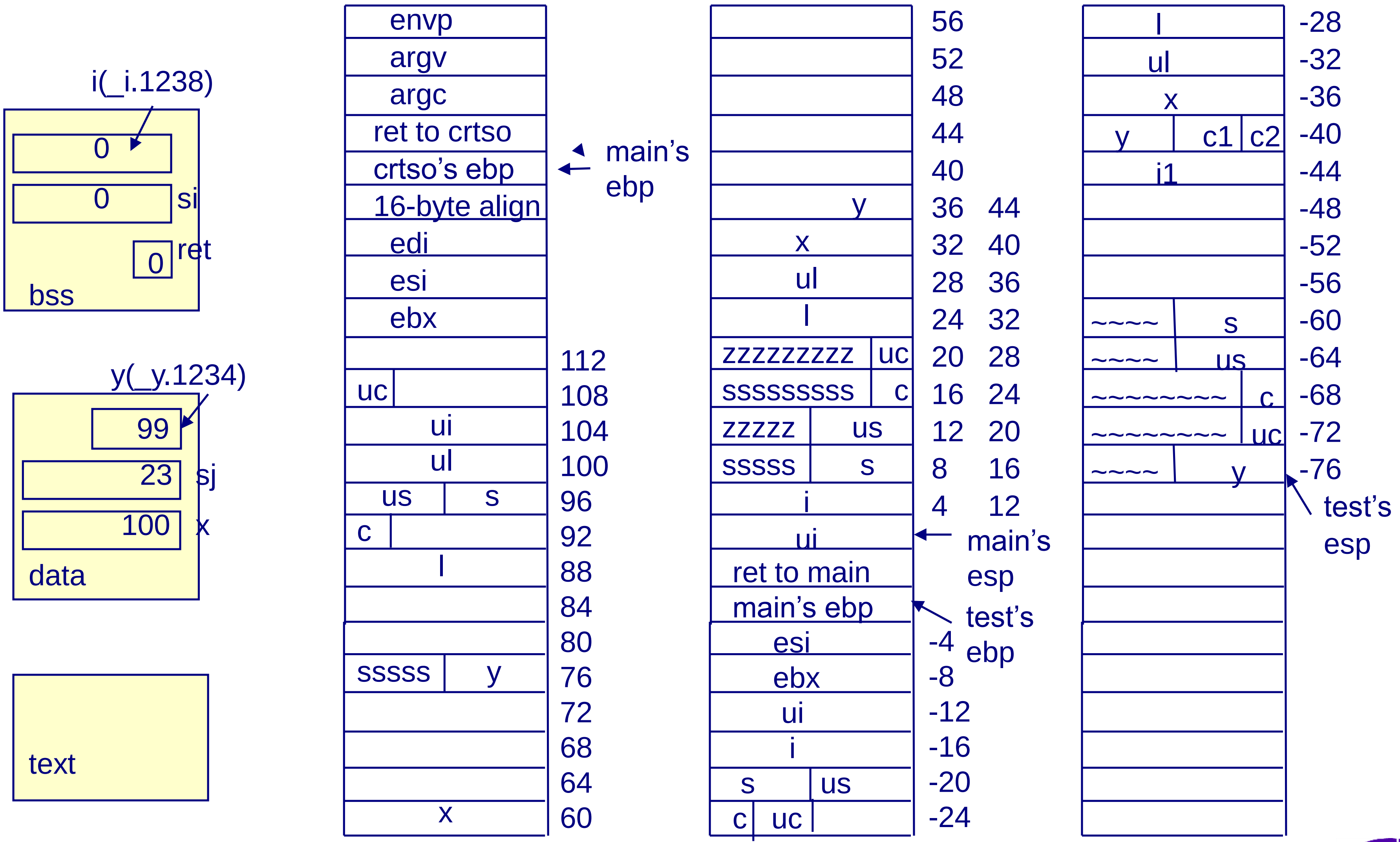
- To generate comments:

  $ gcc –c -Wa,-adhln -g source_code.c > assembly_list.s

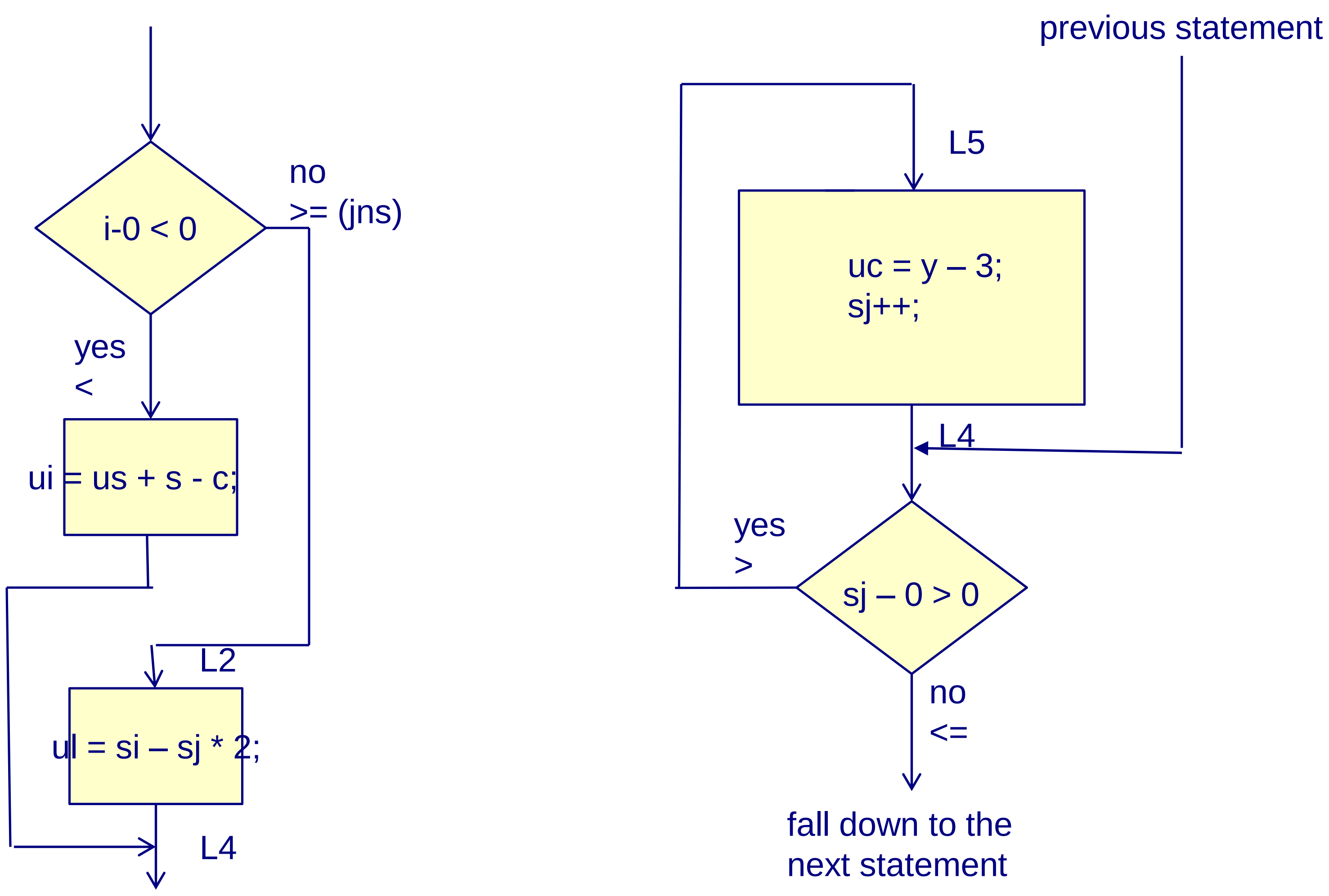
  — -g: Produce debugging information
  — -c: Do not link (to eliminate unnecessary link errors)
  — -Wa,option: Pass option as an option to the assembler (no space between -Wa, and -adhl)
  — -adhln:
  — a: turn on listings
  — d: omit debugging directives; n: omit forms processing
  — h: include high-level source
  — l: include assembly

- However, since compilation with the above options generates too many unnecessary lines, I put comments by myself (by using editors) in the code examples that I provide for the course

Execuion of var.c on Pentium (x_86 gcc on Cygwin)



If and While statements (var.c) (gcc x86)



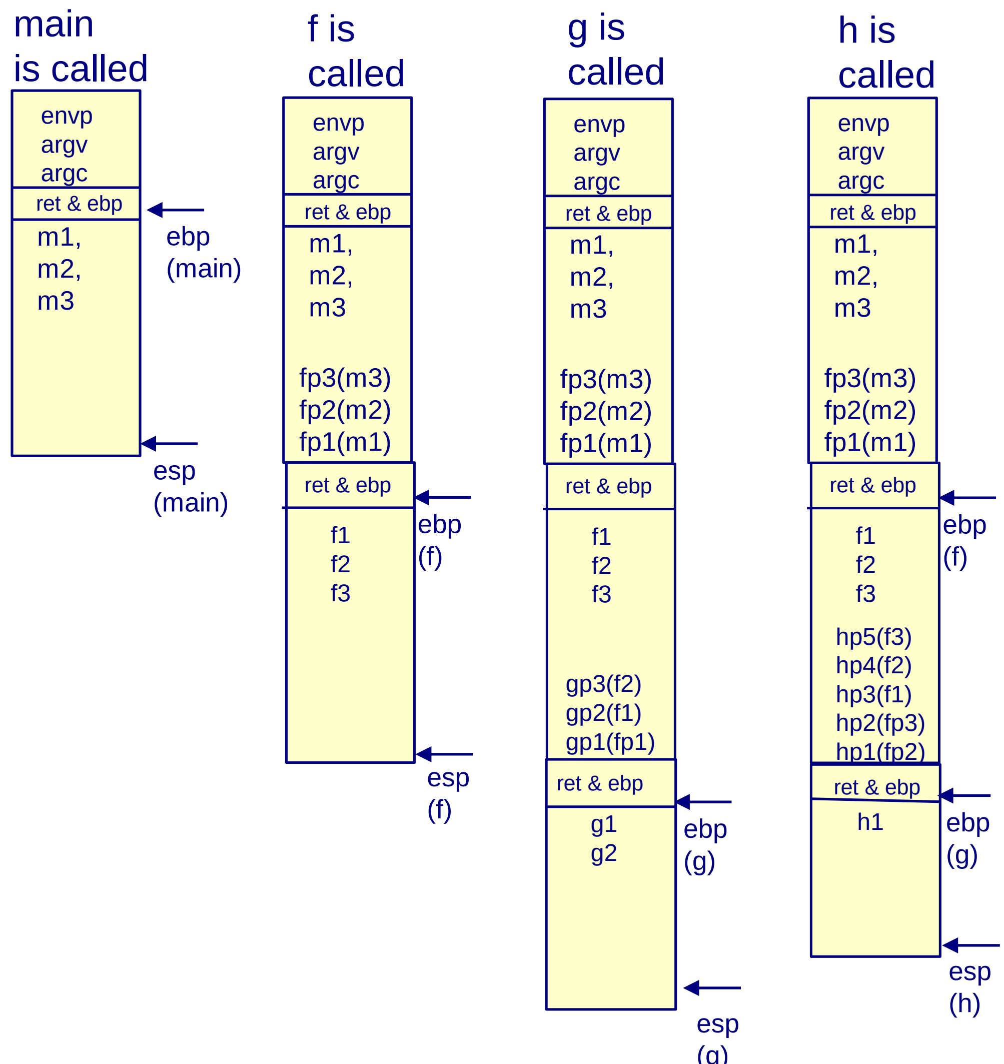
gcc compiler for x86

- Many compilers I know of treat main() as just a regular function.
- However, the gcc compiler I use in the course treat main as a special function:
  1. It calls __main to do necessary initialization of system data structures
  2. It aligns allocation of variables by “andl $-16, $esp”. It does not insert this code in other functions.
  3. Because of the above 2, variables allocated below the alignment filler cannot be accessed via %ebp relative. Therefore, all local variables are accessed via %esp relative
     – In other functions, %ebp is used to access local variables and parameters.
- In each frame, in addition to the areas for (1) parameters, (2) book keeping (area for return address and the caller’s %ebp), and (3) the local variables, the area for parameters of the functions that this function may call is allocated, and %esp points to the bottom of this area.
  – To “push” parameters for callees, “mov” with %esp relative addressing is used, instead of “push” operation

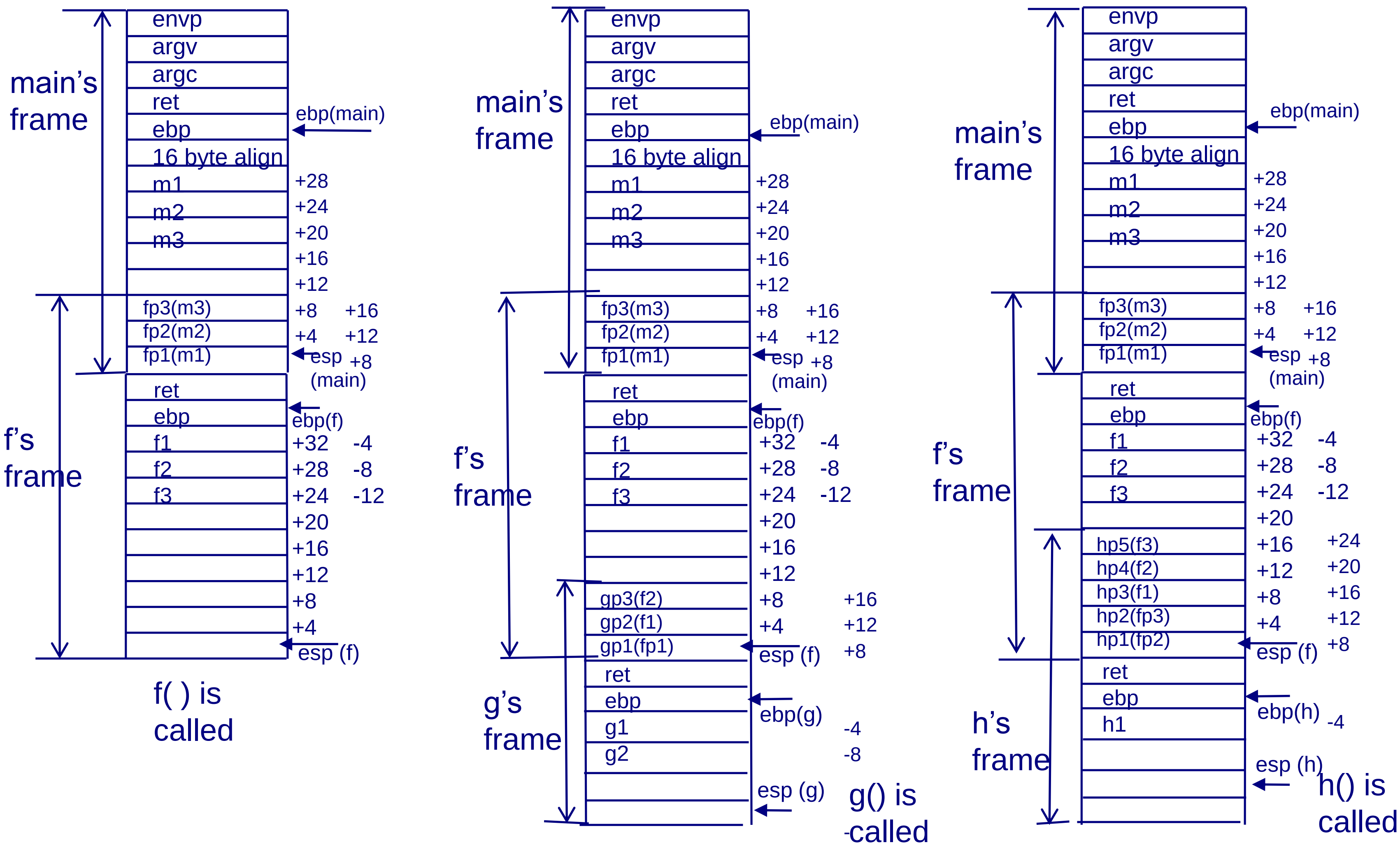
Stack frames created by execution of gcc code (x86)

- Consider the following program:

```
int h(int hp1, int hp2, int hp3,
          int hp4, int hp5) {
   int h1;
        ….
}

int g(int gp1, int gp2, int gp3) {
   int g1, g2;
      ….
}

void f(int fp1, int fp2, int fp3) {
   int f1, t2, f3;
      f1 = g(fp1, f1, f2);
      f2 = h(fp2, fp3, f1, f2, f3);
}

void main() {
     int m1, m2, m3;
     f(m1, m2, m3)
}
```



Params_test_gcc_x86.pdf



Execution Environment of C Programs on Pentium (x86)

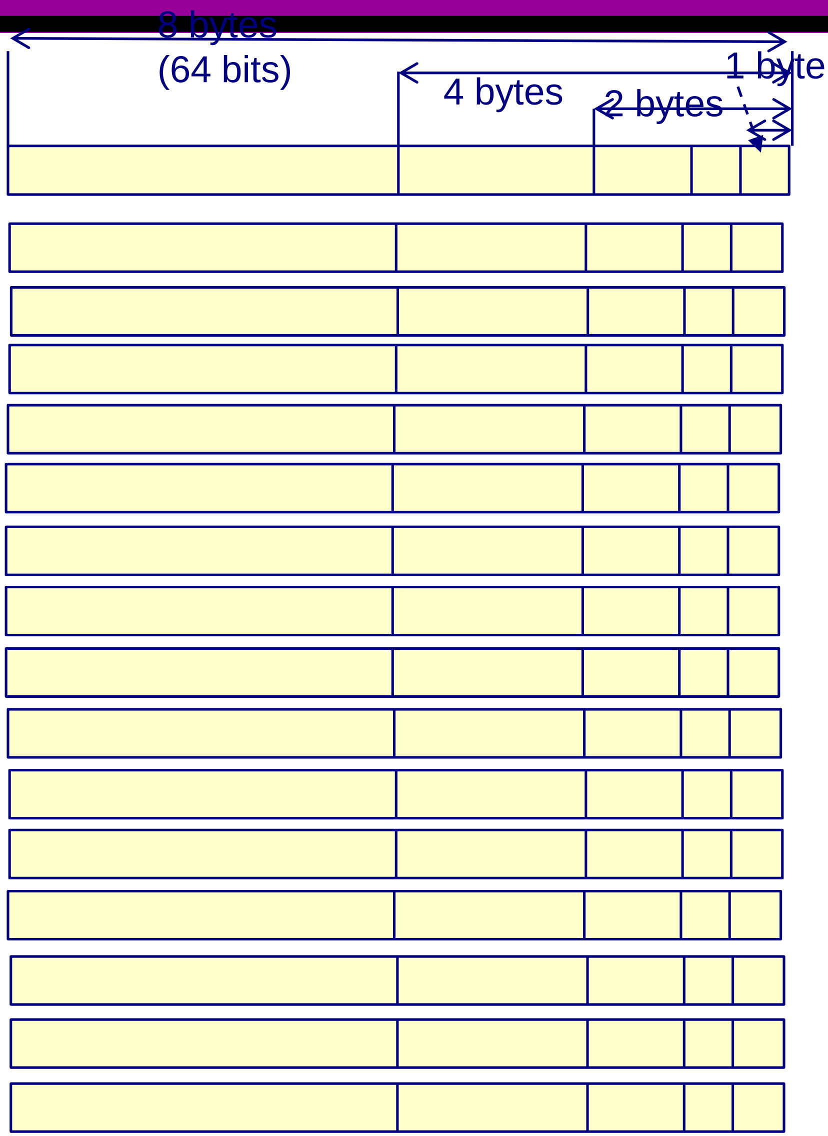
- VC++
- Minix cc
- gcc

- **Execution Environment of C program on 64 bit Core i x64**

Execution Environment of C Programs on H-8

Register usage of x64 (integer register only)

https://msdn.microsoft.com/en-us/library/9z1stfyw.aspx



64-bit 8-byte	lower 32-bit 4-byte	lowest 16-bit	lowest 8-bit	use	destructive/ non-destructive
rax	eax	ax	al	return value	D
rcx	ecx	cx	cl	1st int argument	D
rdx	edx	dx	dl	2nd int argument	D
r8	r8d	r8w	r8b	3rd int argument	D
r9	r9d	r9w	r9b	4th int argument	D
r10	r10d	r10w	r10b		D
r11	r11d	r11w	r11b		D
r12	r12d	r12w	r12b		ND
r13	r13d	r13w	r13b		ND
r14	r14d	r14w	r14b		ND
r15	15d	r15w	r15b		ND
rdi	edi	di	dil		ND
rsi	esi	si	sil		ND
rbx	ebx	bx	bl		ND
rbp	ebp	bp	bpl	frame pointer	ND
rsp	esp	sp	spl	stack pointer	ND

VC++ x64 Data Types

- 1 byte: bool, char (signed char), unsigned char, __int8
- 2 bytes: short, unsigned short, __int16, _wchar_t, __wchar_t
- 4 bytes: int, unsigned int, long, unsigned long, float, __int32
- 8 bytes: double, long long, long double, __int64

  — The above information is from :
    https://msdn.microsoft.com/en-us/library/s3f49ktz.aspx

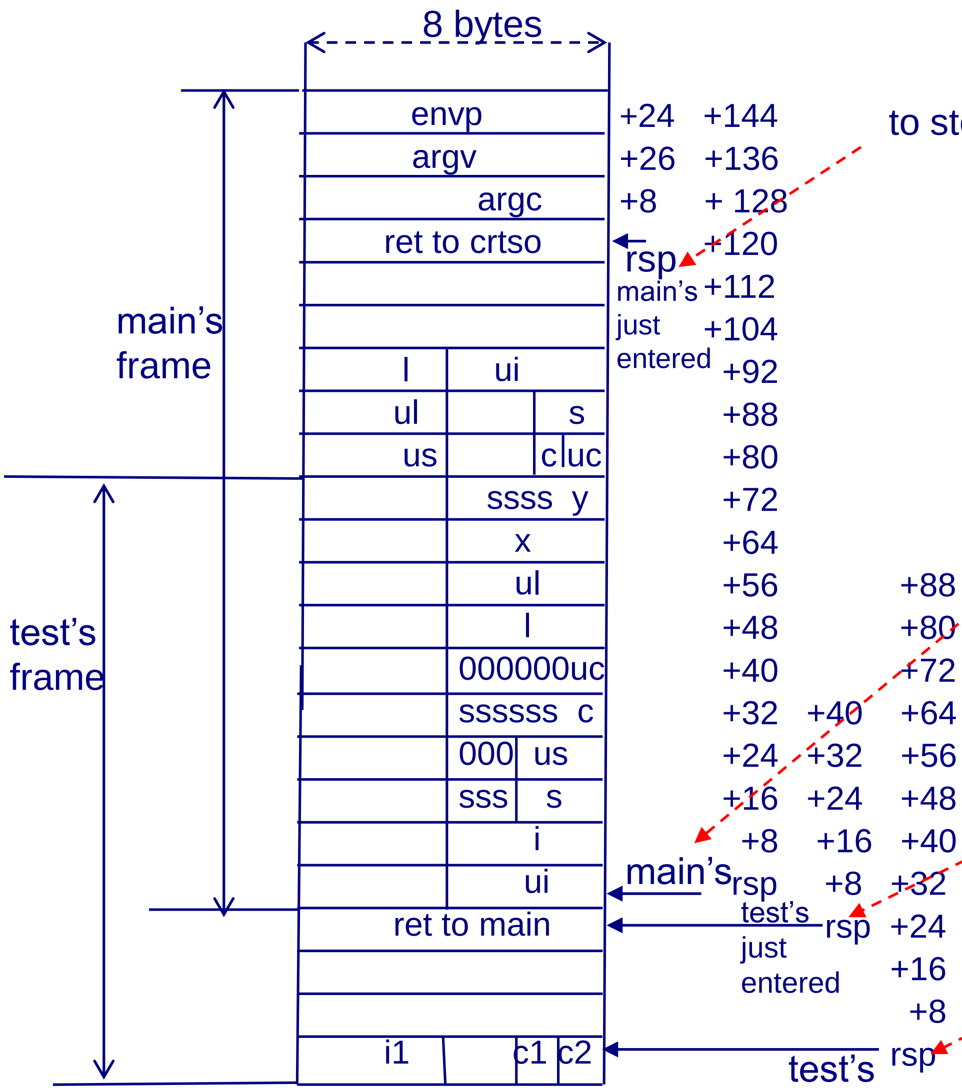
VC++ x64 Calling Convention

- First 4 arguments are passed using registers
  - 1st argument: RCX
  - 2nd argument: RDX
  - 3rd argument: R8
  - 4th argument : R9
  - Space is allocated in the parameter area for those four arguments (called "shadow store") for the callee to save those registers
- The additional arguments are passed by being pushed on the stack frame
  - The above information is from:
    - https://msdn.microsoft.com/en-us/library/ms235286.aspx

VC++ x64 Calling Convention (by my observation)

- Like gcc_x86, a stack frame of a function f() consists of (from top to bottm)
  1. parameters to f( )
  2. return address of the caller
  3. local variables of f( )
  4. area for outgoing parameters of functions called from f( )
     – Therefore, caller's and callee's frames are overlapping each other in the area for outgoing parameters
- Frame pointer (rbp) is not used.
- Stack pointer (rsp) points to the bottom of the frame
  — All variables (above 1, 2, and 4) are accessed by relative to rsp
  — Outgoing parameters of a callee are pushed (more accurately "saved") by the "mov" operation relative to rsp

var.c memory image (VC_x86_64)



to store the first (up to) 4 parameters to main

sp for main's frame:
to access parameters and local variables of main and outgoing parameters from main

to store the first (up to) 4 parameters to test

sp for test's frame:
to access parameters and local variables of main and outgoing parameters from main

Advanced C Techniques for Embedded Systems Programming

Execution Environment of C Programs on Pentium (x86)

- VC++
- Minix cc
- gcc

Execution Environment of C program on 64 bit Pentium x86_6

- Execution Environment of C Programs on H-8

H-8 CPU Register Organization

- H-8 has 16 of 8-bit (or 8 of 16-bit) general purpose registers (R7 is a stack pointer)

7 … 0	7 … 0	16bit
ROH	ROL	R0
R1H	R1L	R1
R2H	R2L	R2
R3H	R3L	R3
R4H	R4L	R4
R5H	R5L	R5
R6H	R6L	R6
R7H (SP)	R7L	R7

PC		
	CCR	Condition Code Register

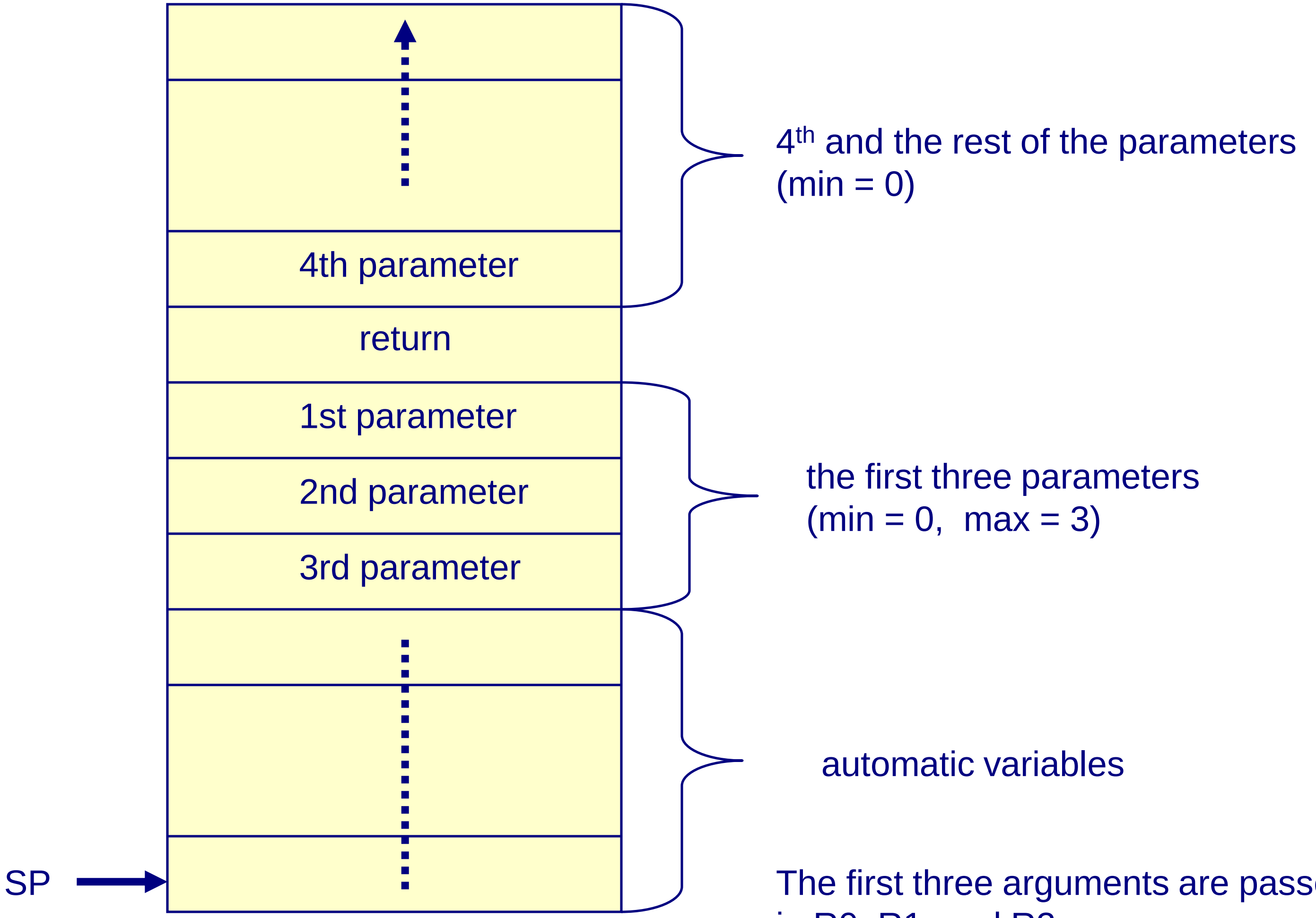
H-8 gcc Compiler

- Code generated by the H-8 gcc compiler
  - there is no frame pointer
    - Parameters and automatic variables are accessed by SP (Stack Pointer) relative addressing
    - Upon preparation for a function call, the offsets of the caller's parameters and automatic variables change as parameters are pushed onto the callee's frame

H-8 Stack Frame Structure (Template)

- The caller places the first three parameters in registers R0, R1, and R2 (the area for these parameters are reserved in the frame) and pushes the rest of the parameters on the Stack frame
- The callee places the first three parameters found in R0, R1, and R2 in the reserved area in the Stack frame; therefore, all parameters are eventually placed on the Stack frame
  - This approach may be typical in the stack frame construction of RISC (Reduced Instruction Set Computer) compilers (or just gcc ??) and also found in the SPARC gcc compiler
  - In the H-8 gcc, the first three and the rest of the parameters are placed in two distinct areas separated by the area for “return address”; furthermore, the first three parameters and the rest of the parameters are ordered in the opposite directions

H-8 Stack Frame Structure (Template) (cont)



The first three arguments are passed in R0, R1, and R2

Return Value of Function in H-8

- In gcc for H-8, the return value of a function is passed back to the caller by using the following register(s):
  - 2 bytes : R0
  - 4 bytes : R0 (MSW) + R1 (LSW) pair

H-8 gcc Assembler Convention

- Pseudo instructions of H-8 gcc
  - .global： export a label
    - .global _x
  - .comm： allocate a specified number of bytes in the bss segment, associate a label with the address, and export the label
    - .comm _ret, 1
  - .lcomm： (local communal) allocate a specified number of bytes and associate a lable with the address, but do not export the label
    - .lcomm _si, 2
  - .word： allocate 2 bytes in the data segment and initialize the area with the given value
    - .word 23
  - #VALUE： represent immediate data (no memory area is allocated; the value is embedded in the instruction)
    - #2
  - @ADDRESS： represent the contents of the memory at "ADDRESS"
    - mov.w @(4, r7), r2
    - mov.w r2, @_ans

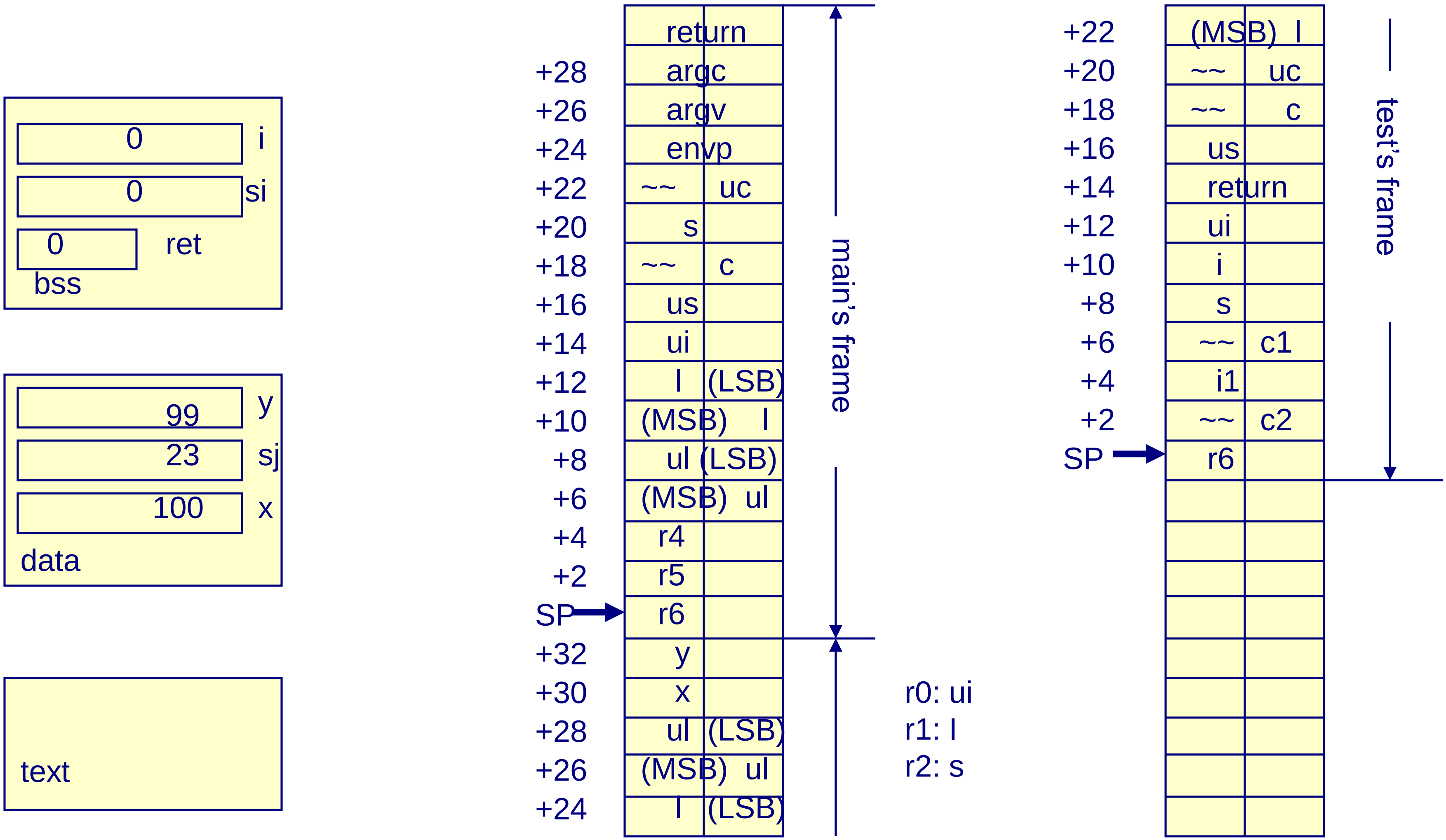
H-8 gcc Assembler Convention (cont)

- Assembler format
  - operation src, dest (in subtract operation, dest = dest-src)

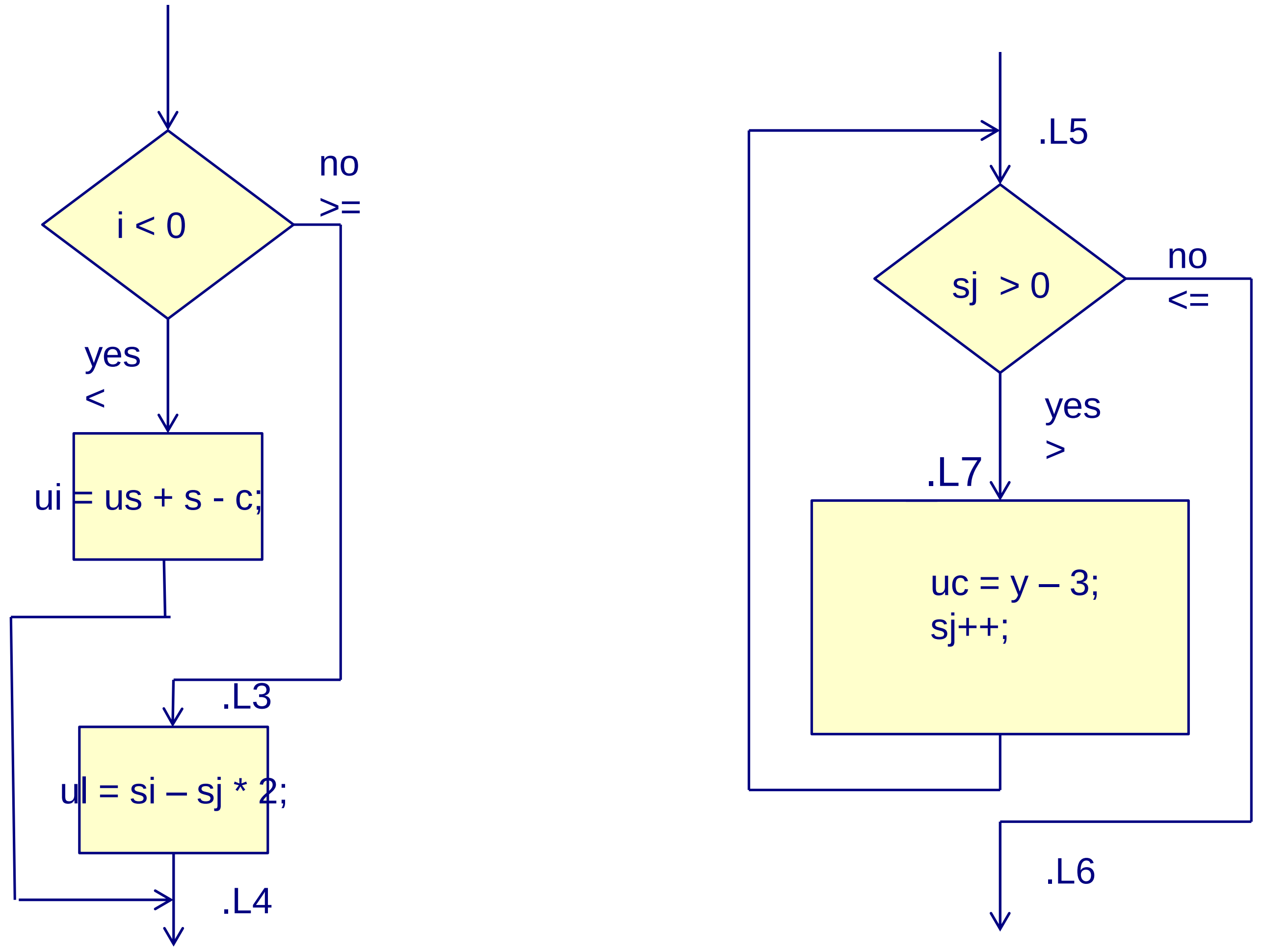
H-8 destructive and non-destructive registers

- The gcc compiler for H-8 manages R0~R3 to be destructive and R4, R5 to be non-destructive registers
  - R7 is a stack pointer
  - This version of compiler saves R6 in each function (except for main in fact.c) no matter whether it is used in the function. I do not know the reason
    - More recent versions of the gcc compiler (i.e., ver 3.4.3 used in this course) do not save R6
- main() in var.c uses R4 and R5 to push parameters; therefore, it saves R4 and R5 (and R6 as mentioned above) at the entry to main and restores them at the end
  - The ver 3.4.3 compiler does not use R4 or R5 in main. Therefore, it does not save/restore any register in main
- The goal of this course is NOT to understand a particular version of a particular compiler in detail. The goal is to obtain knowledge that is applicable to any version of any compiler for any processor. Therefore, we study code generated by an older version of the compiler since we can actually observe the save/restore process for the non-destructive registers.

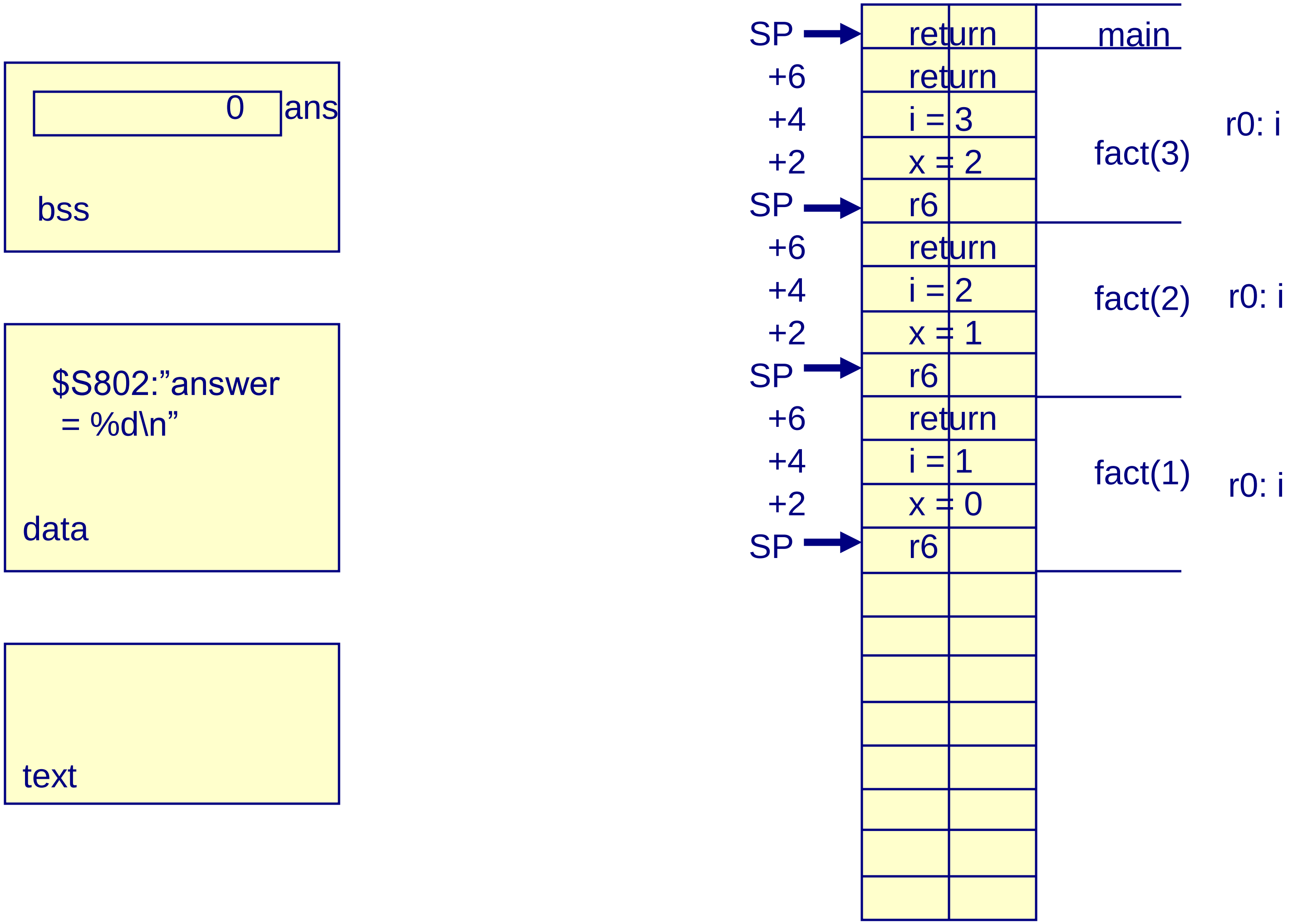
Execution of var.c on H-8 (gcc)



If and While Statements (var.c) (H-8 gcc)



Execution of fact.c on H-8 (gcc)



short cut evaluation (fact.c) (H-8 gcc)