Advanced C Techniques for Embedded Systems Programming

K·STATE
Kansas State University

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)

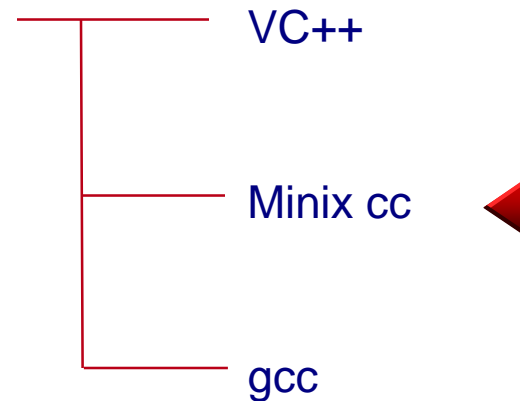   _si   DD   01H DUP (?)   // for "static int si;"

            // 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 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

- 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
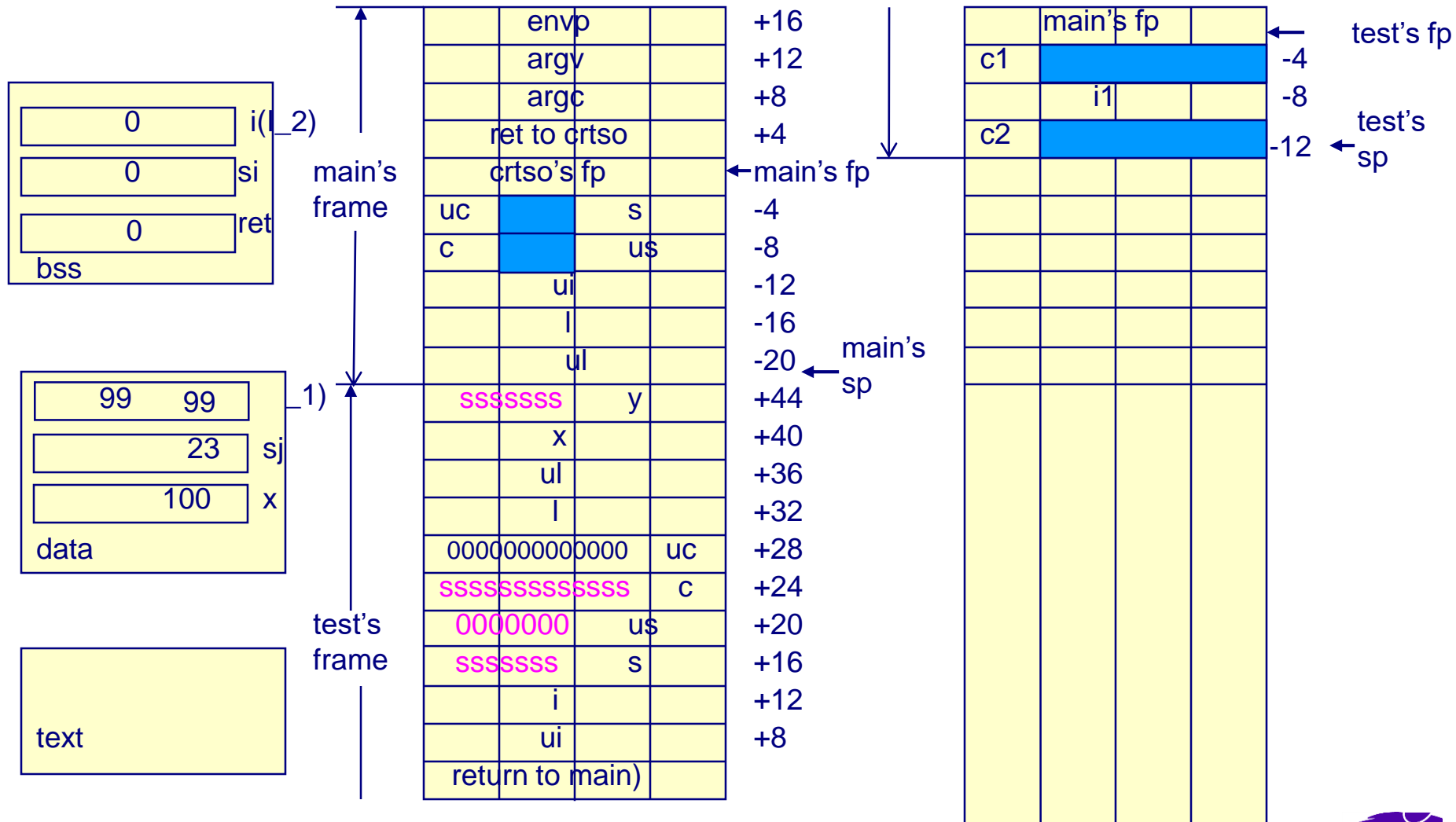 - int x = 100; is compiled to
    .export _x
    _x:
    .data4 100

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

bss

0	i(l_2)
0	si
0	ret

data

99	99
23	sj
100	x

text

main's frame

envp				+16
argv				+12
argc				+8
ret to crtso				+4
crtso's fp				←main's fp
uc		s		-4
c		us		-8
ui				-12
l				-16
ul				-20 ← main's sp

test's frame

sssssss		y		+44
x				+40
ul				+36
l				+32
0000000000000		uc		+28
sssssssssssss		c		+24
0000000		us		+20
sssssss		s		+16
i				+12
ui				+8
return to main)				

main's fp			← test's fp
c1			-4
	i1		-8
c2			-12 ← test's sp

```
                                    no
                                    >=
              i-0 < 0
                 |
               yes
                <
                 |
          ui = us + s - c;
                 |
                 |                  I1_4
          ul = si – sj * 2;
                 |
                                    I1_7
```

```
                                    I1_7
              sj – 0 > 0            no
                 |                  <=
               yes
                >
                 |
          uc = y – 3;
          sj++;
                 |
                                    I1_6
```
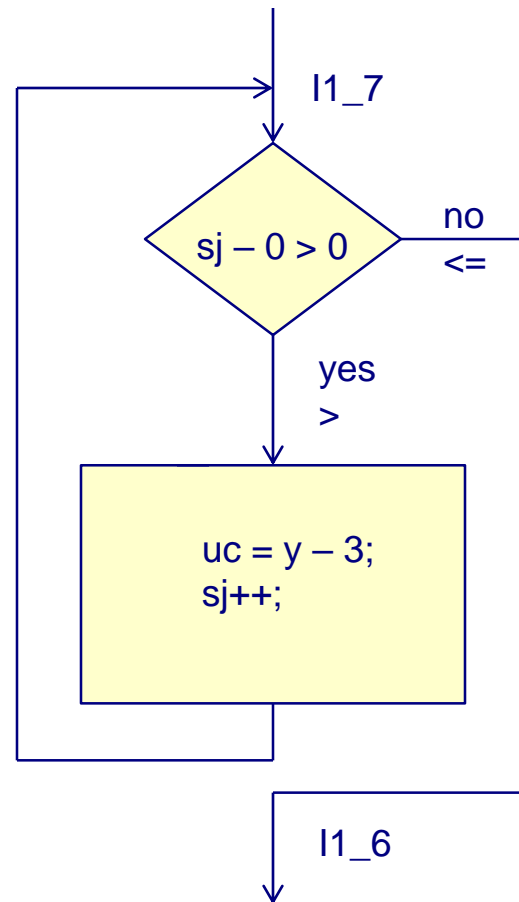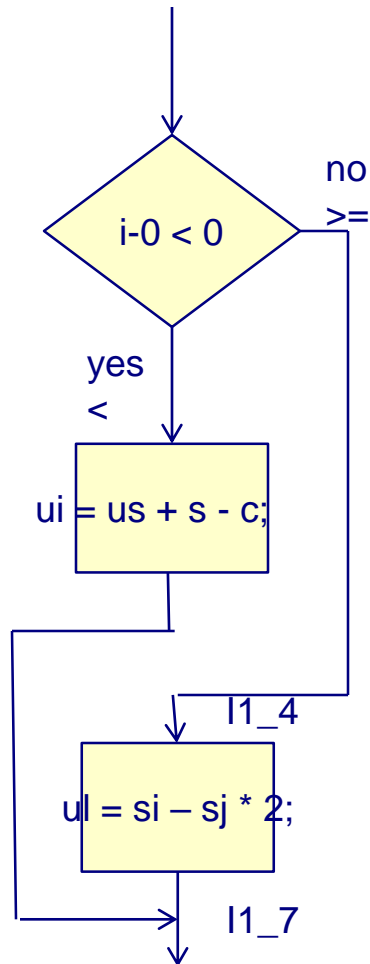
Execution of fact.c on Pentium (Minix cc)

		envp		+16	
		argv		+12	
main's frame		argc		+8	
		ret to crtso		+4	
		crtso's fp		←main's fp	
		i (3)		+8	
fact(3)'s frame		ret to main		+4	
		main's fp		←fact(3)'s fp	
		esi		-4	x: esi = 2
		i (2)		+8	
fact(2)'s frame		ret to fact		+4	
		fact(3)'s fp		← fact(2)'s fp	
		esi		-4	x: esi = 1
		i (1)		+8	
fact(1)'s frame		ret to fact		+4	
		fact(2)'s fp		←fact(1)'s fp	
		esi		-4	x: esi = 0

bss
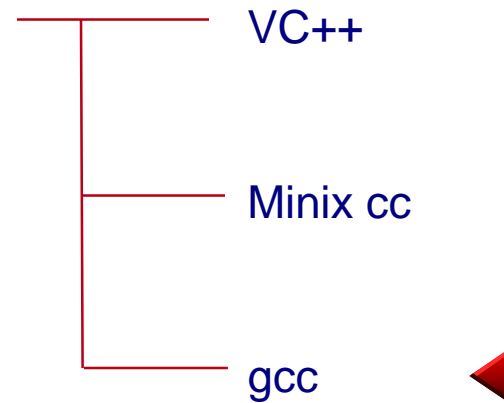
0 ans

data

rom

l_1:"answer = %d\n"

text
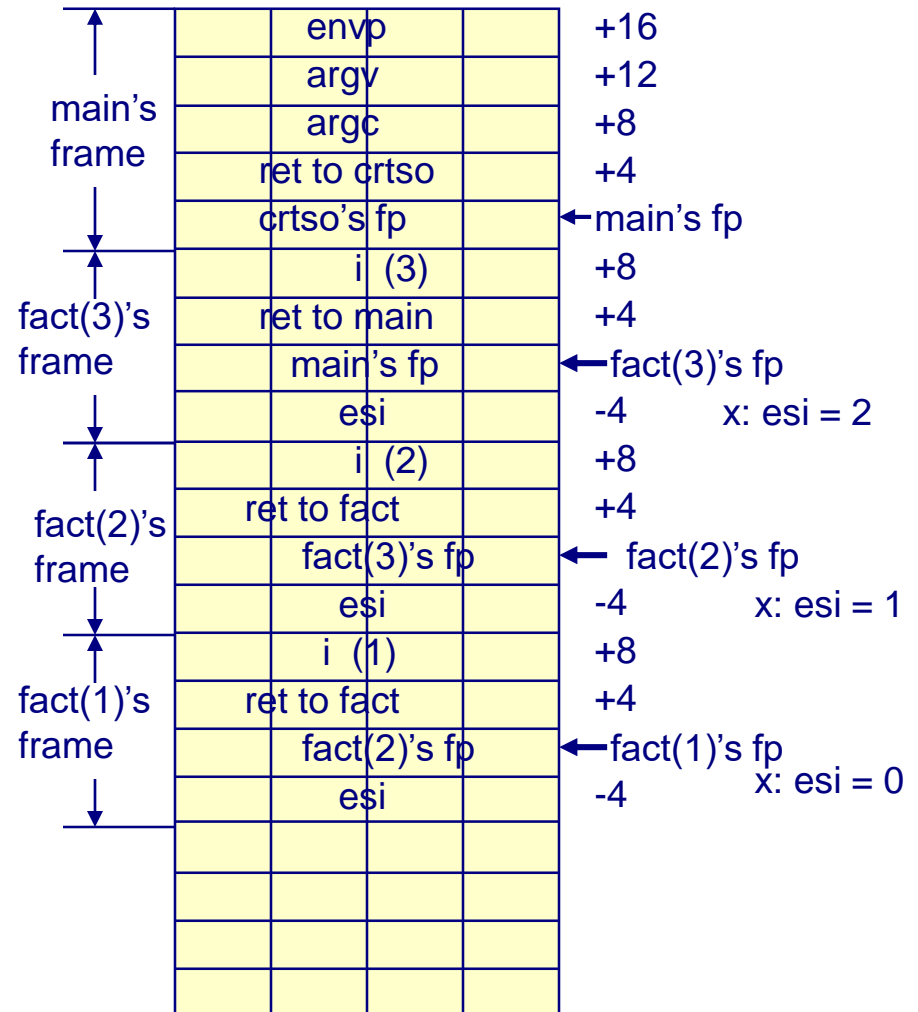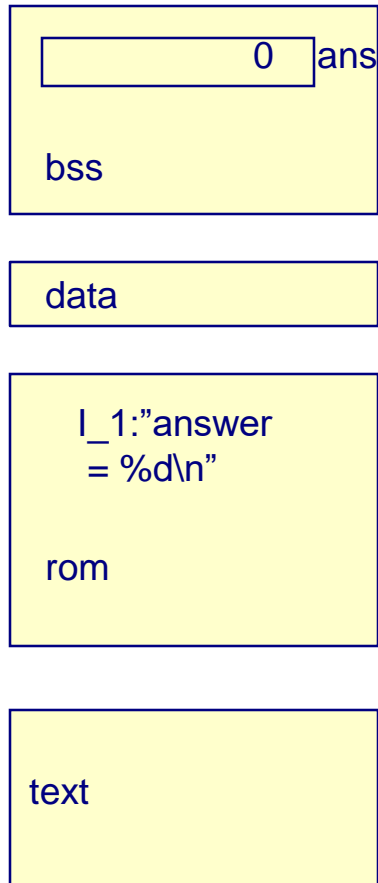
short cut evaluation (fact.c) (Minix cc)
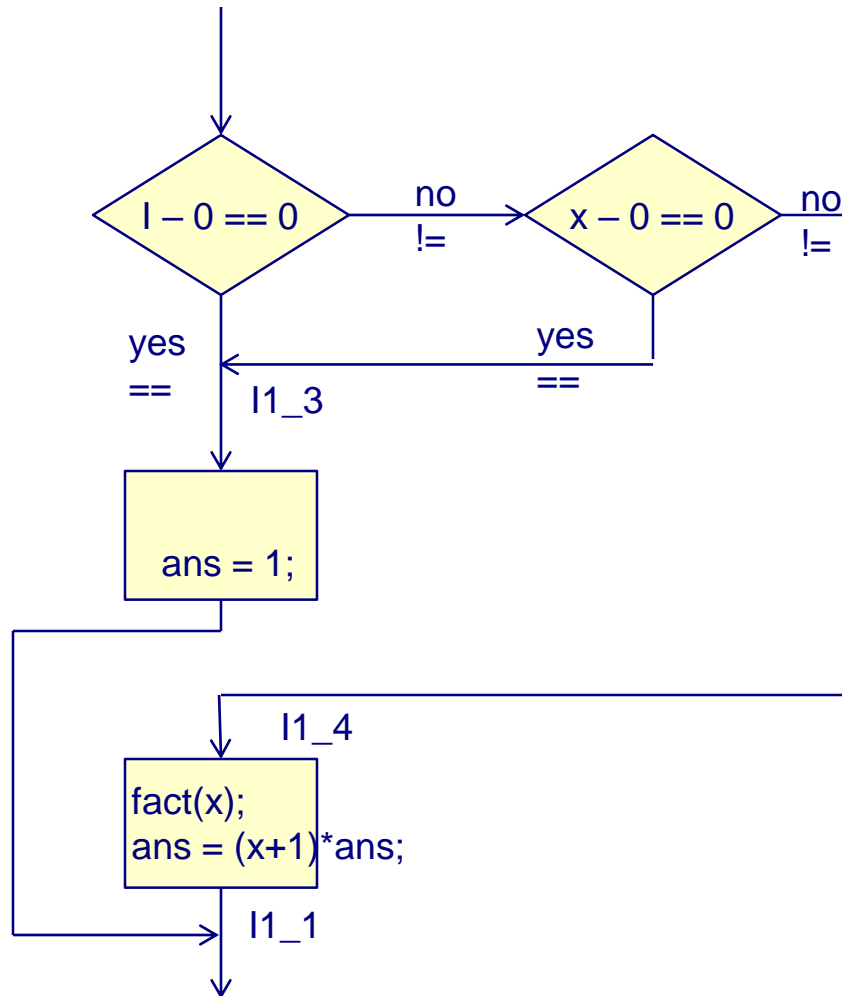
Advanced C Techniques for Embedded Systems Programming

▶ Execution Environment of C Programs on Pentium (x86)

VC++

Minix cc

gcc ◀

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

i(_i.1238)

bss

0
0 si
0 ret

y(_y.1234)

data

99
23 sj
100 x

text
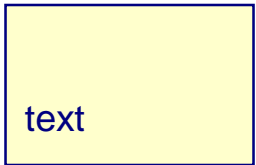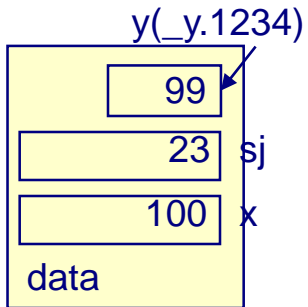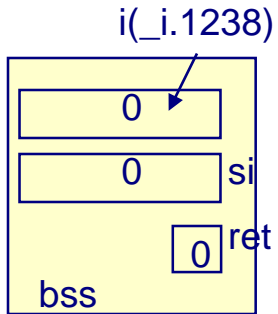
envp	
argv	
argc	
ret to crtso	
crtso's ebp	← main's ebp
16-byte align	
edi	
esi	
ebx	
	112
uc	108
ui	104
ul	100
us s	96
c	92
l	88
	84
	80
sssss y	76
	72
	68
	64
x	60

	56	
	52	
	48	
	44	
	40	
y	36	44
x	32	40
ul	28	36
l	24	32
zzzzzzzz uc	20	28
ssssssss c	16	24
zzzzz us	12	20
sssss s	8	16
i	4	12
ui	← main's esp	
ret to main		
main's ebp		test's ebp
esi	-4	
ebx	-8	
ui	-12	
i	-16	
s us	-20	
c uc	-24	

l	-28
ul	-32
x	-36
y c1 c2	-40
i1	-44
	-48
	-52
	-56
~~ s	-60
~~ us	-64
~~ c	-68
~~ uc	-72
~~ y	-76

test's esp

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

previous statement

no
>= (jns)

i-0 < 0

yes
<

ui = us + s - c;

L2

ul = si – sj * 2;

L4

L5

uc = y – 3;
sj++;

L4

yes
>

sj – 0 > 0

no
<=

fall down to the
next statement

CIS621K-3-18
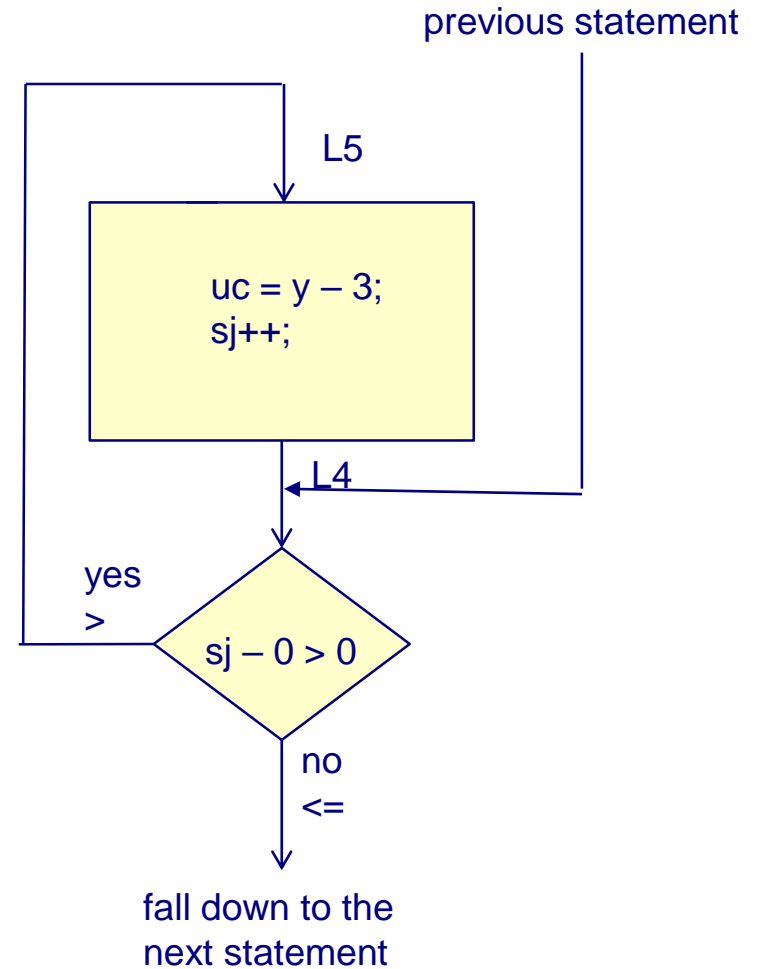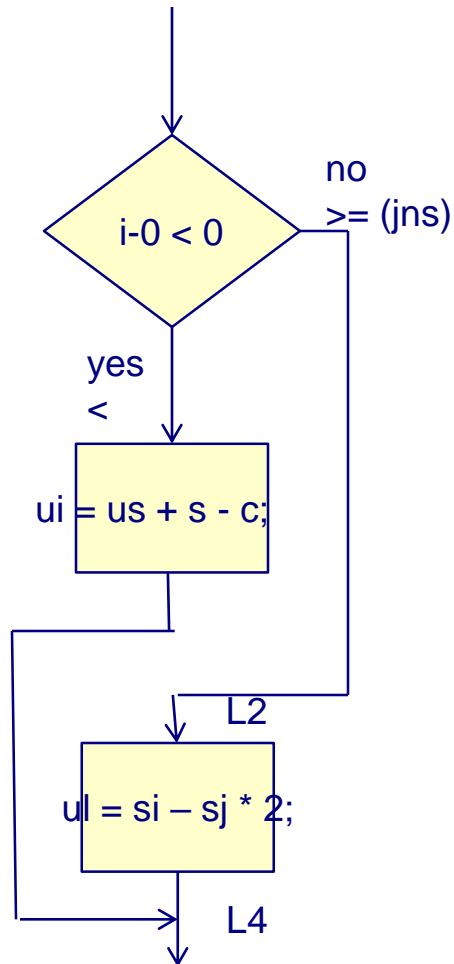
- Many compilers I know of treat main() as just a regular function.

- 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 align allocation of variables by "andl $-16, $esp". It does not insert this code in other funtions.
  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.
  4. 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)
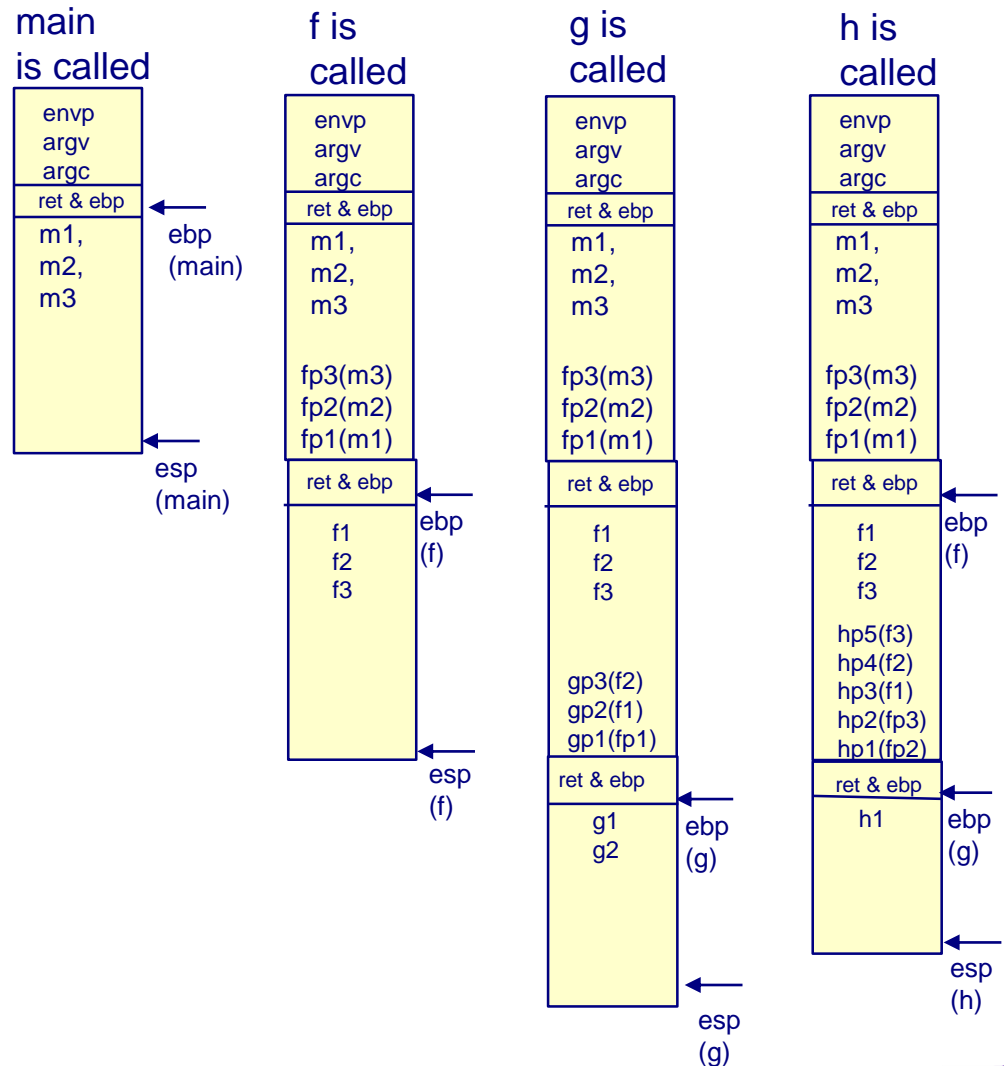
- **Consider the following program:**

```
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);
}
```

main is called

envp argv argc
ret & ebp
m1, m2, m3

← ebp (main)

← esp (main)

f is called

envp argv argc
ret & ebp
m1, m2, m3
fp3(m3) fp2(m2) fp1(m1)
ret & ebp
f1 f2 f3

← ebp (main)

← ebp (f)

← esp (f)

g is called

envp argv argc
ret & ebp
m1, m2, m3
fp3(m3) fp2(m2) fp1(m1)
ret & ebp
f1 f2 f3
gp3(f2) gp2(f1) gp1(fp1)
ret & ebp
g1 g2

← ebp (g)

← esp (g)

h is called

envp argv argc
ret & ebp
m1, m2, m3
fp3(m3) fp2(m2) fp1(m1)
ret & ebp
f1 f2 f3
hp5(f3) hp4(f2) hp3(f1) hp2(fp3) hp1(fp2)
ret & ebp
h1

← ebp (f)

← ebp (g)

← esp (h)

Column 1 — f( ) is called

Label	Offset	Offset 2
envp		
argv		
argc		
ret		
ebp		
16 byte align	ebp(main)	
m1	+28	
m2	+24	
m3	+20	
	+16	
	+12	
fp3(m3)	+8	+16
fp2(m2)	+4	+12
fp1(m1)	esp +8 (main)	
ret		
ebp	ebp(f)	
f1	+32	-4
f2	+28	-8
f3	+24	-12
	+20	
	+16	
	+12	
	+8	
	+4	
	esp (f)	

f( ) is called

Column 2 — g() is called

Label	Offset	Offset 2
envp		
argv		
argc		
ret		
ebp		
16 byte align	ebp(main)	
m1	+28	
m2	+24	
m3	+20	
	+16	
	+12	
fp3(m3)	+8	+16
fp2(m2)	+4	+12
fp1(m1)	esp +8 (main)	
ret		
ebp	ebp(f)	
f1	+32	-4
f2	+28	-8
f3	+24	-12
	+20	
	+16	
	+12	
gp3(f2)	+8	+16
gp2(f1)	+4	+12
gp1(fp1)	esp (f)	+8
ret		
ebp	ebp(g)	
g1		-4
g2		-8
		-
	esp (g)	

g() is called

Column 3 — h() is called

Label	Offset	Offset 2
envp		
argv		
argc		
ret		
ebp		
16 byte align	ebp(main)	
m1	+28	
m2	+24	
m3	+20	
	+16	
	+12	
fp3(m3)	+8	+16
fp2(m2)	+4	+12
fp1(m1)	esp +8 (main)	
ret		
ebp	ebp(f)	
f1	+32	-4
f2	+28	-8
f3	+24	-12
	+20	
hp5(f3)	+16	+24
hp4(f2)	+12	+20
hp3(f1)	+8	+16
hp2(fp3)	+4	+12
hp1(fp2)	esp (f)	+8
ret		
ebp	ebp(h)	-4
h1		
	esp (h)	

h() is called

▶ Execution Environment of C Programs on Pentium (x86)

VC++

Minix cc

gcc

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

4th and the rest of the parameters
(min = 0)

4th parameter

return

1st parameter

2nd parameter

3rd parameter

the first three parameters
(min = 0, max = 3)

automatic variables

SP

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
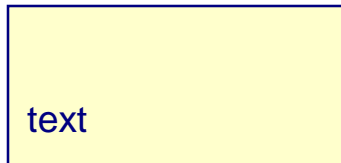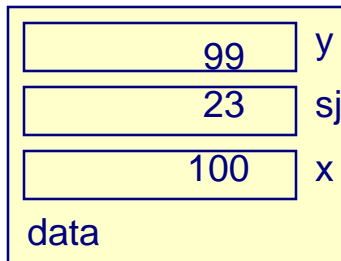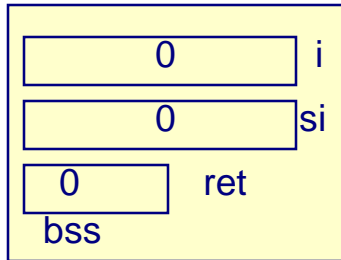 — operation src, dest (in subtract operation, dest = dest-src)

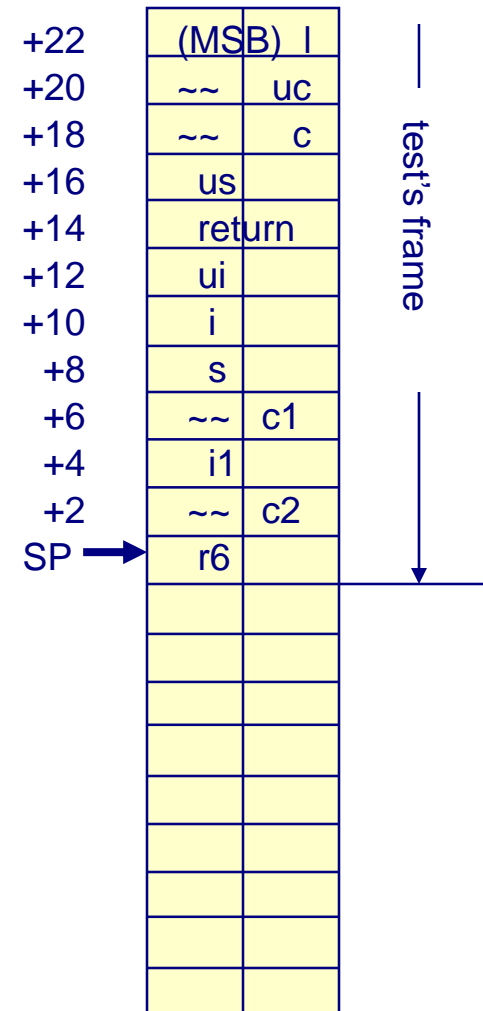# H-8 destructive and non-destructive registers

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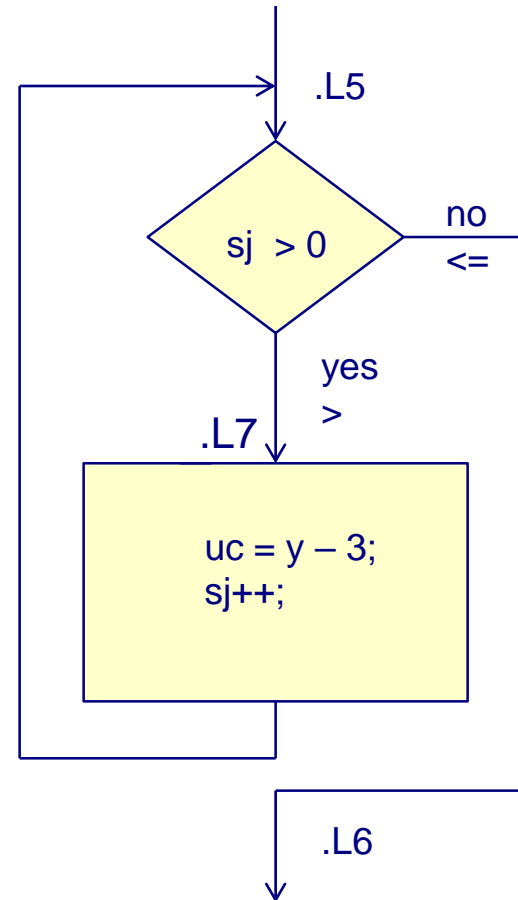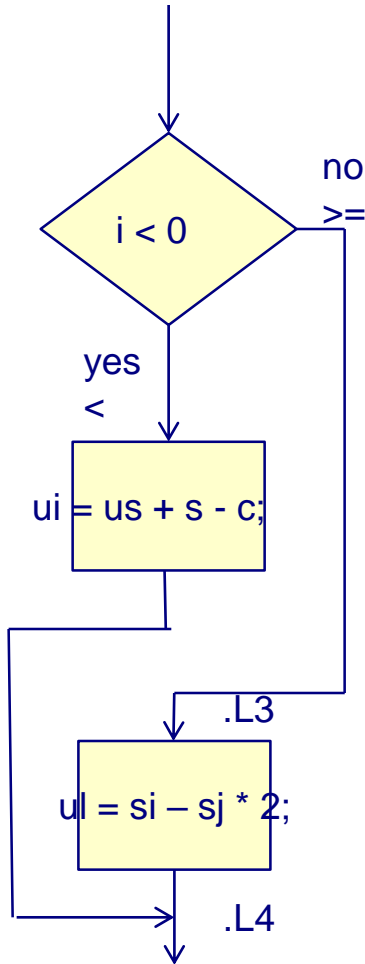
Execution of var.c on H-8 (gcc)

bss

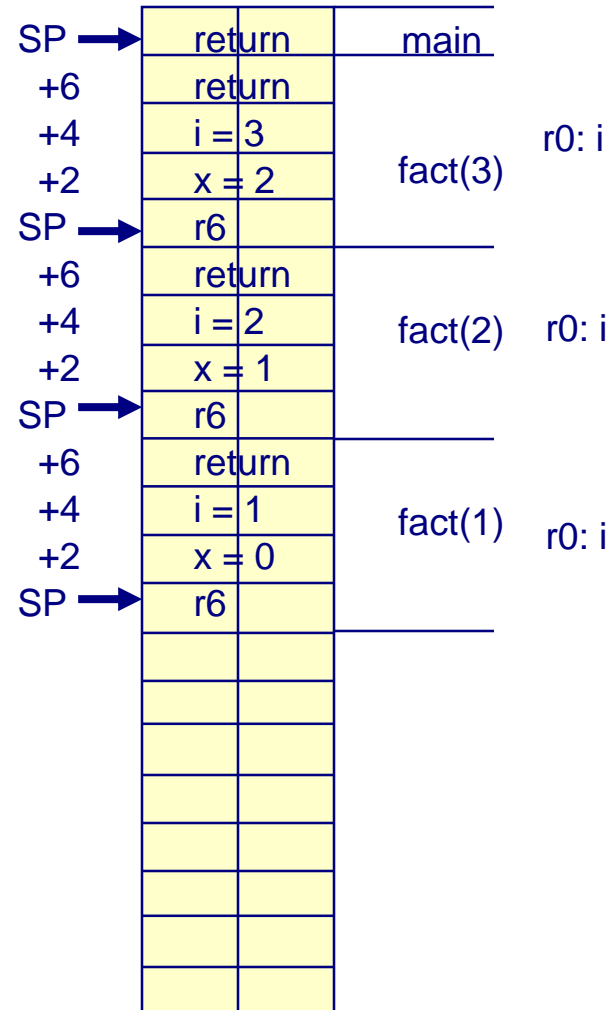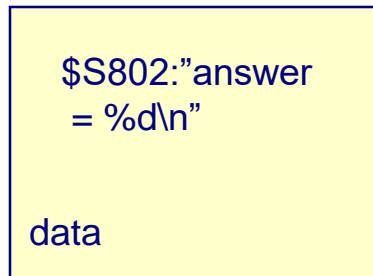|	
0	i
0	si
0	ret

data

99	y
23	sj
100	x

text

main's frame

offset		
	return	
+28	argc	
+26	argv	
+24	envp	
+22	~~	uc
+20	s	
+18	~~	c
+16	us	
+14	ui	
+12	l	(LSB)
+10	(MSB)	l
+8	ul (LSB)	
+6	(MSB) ul	
+4	r4	
+2	r5	
SP →	r6	
+32	y	
+30	x	
+28	ul	(LSB)
+26	(MSB)	ul
+24	l	(LSB)

r0: ui
r1: l
r2: s

test's frame

offset		
+22	(MSB)	l
+20	~~	uc
+18	~~	c
+16	us	
+14	return	
+12	ui	
+10	i	
+8	s	
+6	~~	c1
+4	i1	
+2	~~	c2
SP →	r6	

no
>=

i < 0

yes
<

ui = us + s - c;

.L3

ul = si – sj * 2;

.L4

.L5

no
<=

sj > 0

yes
>

.L7

uc = y – 3;
sj++;

.L6

bss

0 ans

$S802:"answer = %d\n"

data

text

SP →	return	main
+6	return	
+4	i = 3	
+2	x = 2	fact(3) r0: i
SP →	r6	
+6	return	
+4	i = 2	fact(2) r0: i
+2	x = 1	
SP →	r6	
+6	return	
+4	i = 1	fact(1)
+2	x = 0	r0: i
SP →	r6	

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