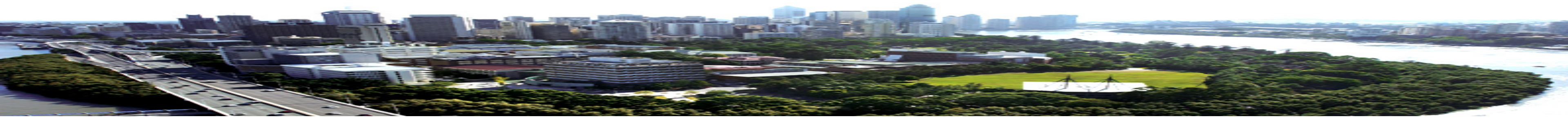
ENB 350 Real-time Computer based systems

Lecture 4

MIXING C AND ASSEMBLY

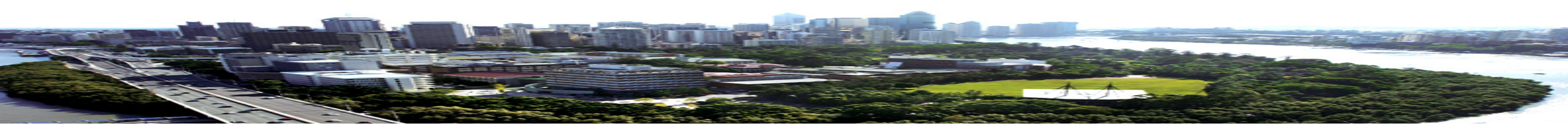
V. Chandran



Contents

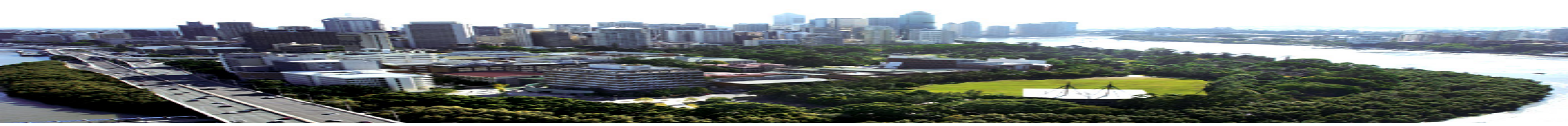
- Interfacing C and assembly
- Passing parameters and returned values
- Local variables
- Stack Frames
- Built-in Routines for Accessing Ports

Read the Dynamic C user manual. It is a good idea to brush up on C. Chapter 12 covers the use of assembly language.



Why assembly?

- May need to access machine internals – specific registers
- Efficient usage of architecture – speed, memory



Rabbit Dynamic C - Assembler format

Operation Field

Contents of memory location whose address is in BC

L1: :

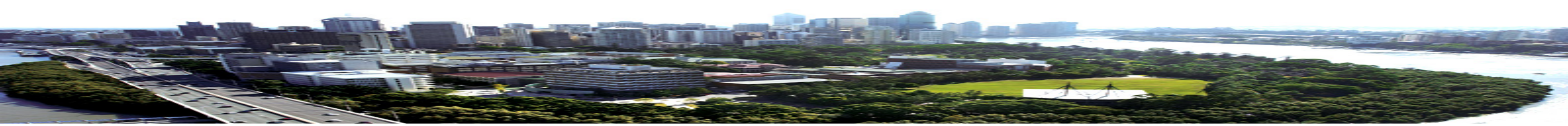
<PREFIX> LD A, (BC); MOVE DATA INTO A

Global scope ::

Label Field

Operand Fields

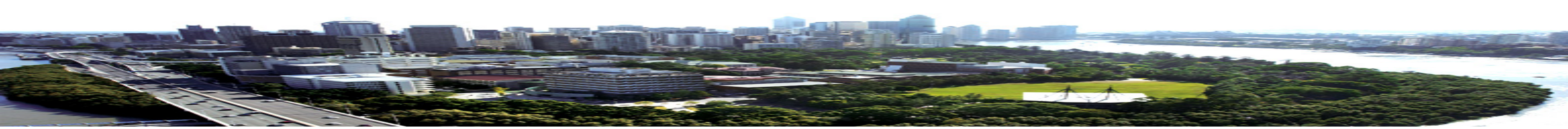
Comment Field



Instruction prefixes

- Rabbit microprocessors have two I/O spaces: internal I/O registers and external I/O registers.
- Prefix IOI -> accesses internal I/O register instead of memory
- Prefix IOE -> accesses external I/O port instead of memory. Full 16 bit address is used.

```
ioi    ld  (SACR),a
```



Register usage

- PC is program counter.
- SP is the stack pointer
- CALL and RET use PC, SP
- DJNZ uses B for a counter
- A is the 8 bit accumulator
- HL is the 16 bit accumulator
- HL and BCDE are used for 16 bit and 32 bit parameters

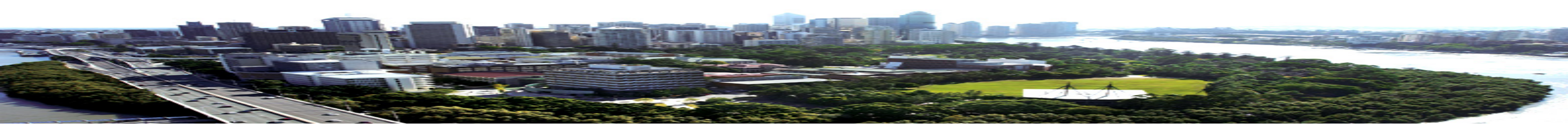
- For accessing extended memory LCALL uses XPC (memory organization will be discussed later)

Function Call and Return

- CALL instruction used to invoke a function
  - Pushes the return address onto the stack.
- RET instruction used in the function to return to the caller.
  - Pops the return address off the stack.

Assembly code with Dynamic C

- Assembly language statements can be embedded in Dynamic C
- Entire functions may be written in assembly language
- C statements can be embedded within assembly code
- C language variables may be accessed by the assembly code
- Parameters may be passed between C language code and assembly code



Inline assembly

```
printf("\n warning light on \n");
#asm
   ld    a, (PBDRshadow)
   set  6,a
   ld    (PBDRshadow), a
ioi ld  (PBDR), a
#endasm
get_status();
```

Compiler directive at the beginning
And one at the end

Get contents of shadow register into A. Set bit 6.

Update shadow register
Then write data to data register

Back to C

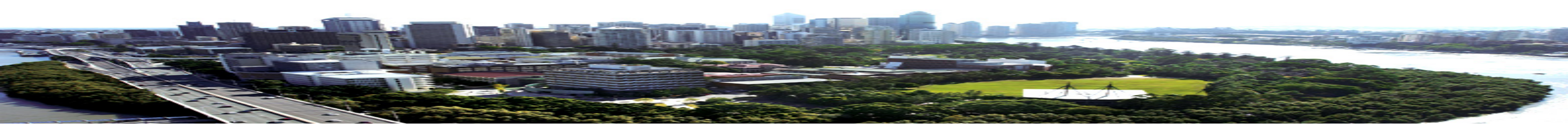
C calling stand-alone assembly

```
int crc_lookup (int  value);

main()
{
   int i,j;
   i=1;
   j=crc_lookup(i);
}

#asm
crc_lookup::
   ….
   ld hl,a
ret
#endasm
```

Function prototype must be declared.

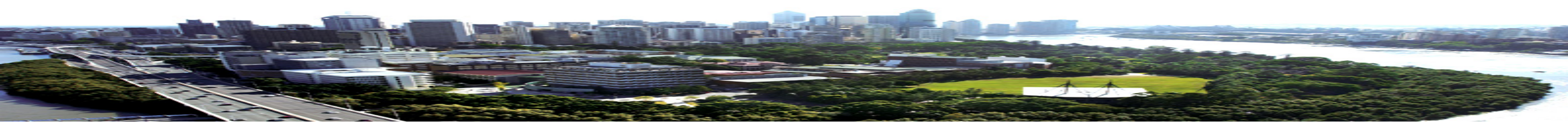
Label same as function name.
Use of double colon declares global scope for the label
Parameter i will be in hl

Return value put in hl before ret.
CPU registers used by assembly need not be saved for functions upon entering or restored on exit. For ISRs, they must be saved on entry and restored on exit.



1 parameter passed and one value returned

```
int result (int x);

// int result (int x)
// the assembly subroutine returns sum of x + 1

#asm root
result::
        // HL contains parameter x

        inc hl              ; x = x+1
        ret                 ; pass x back through HL
#endasm
```



Assembly code implementing a C function

```
int circshift(int x)
{
#asm
ex BC, HL                // parameter x moved from HL
RLC BC
ex BC,HL                 // return value placed in HL
#endasm
}
```

C within assembly

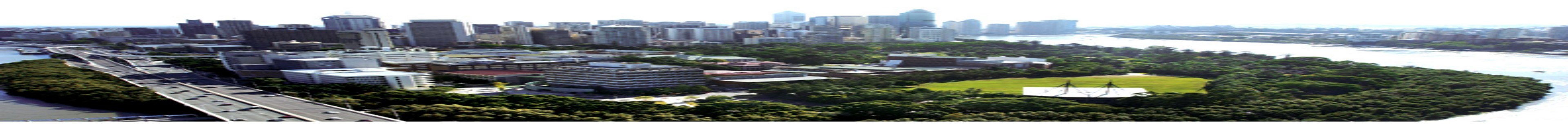
If c statements are to be embedded in the assembly function, they are prefixed by a ‘c’

```
#asm
InitValues::
c start_time = 0;
c counter = 256;
ret
#endasm
```

C variables (global) can be accessed within assembly by

```
ld    a,(temp3)
```

for example where temp3 is an int.



Multiple parameters passed

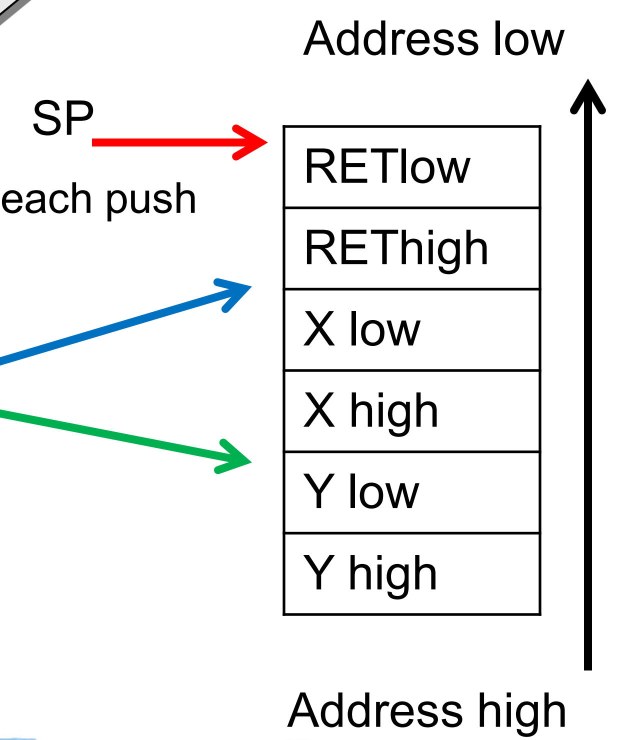
- If the first argument has one or two bytes, only the first argument is pushed into HL (H has the most significant byte)
- If the first argument contains four bytes, the first argument is pushed in to BC:DE (with B containing the most significant byte)
- All arguments, including the first are pushed on to the stack. The last argument is pushed first.
- Returned values are placed in the same manner into HL or BC:DE.

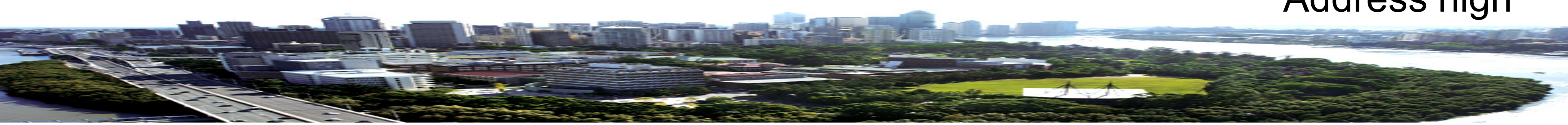
2 parameters example

And so on for more parameters

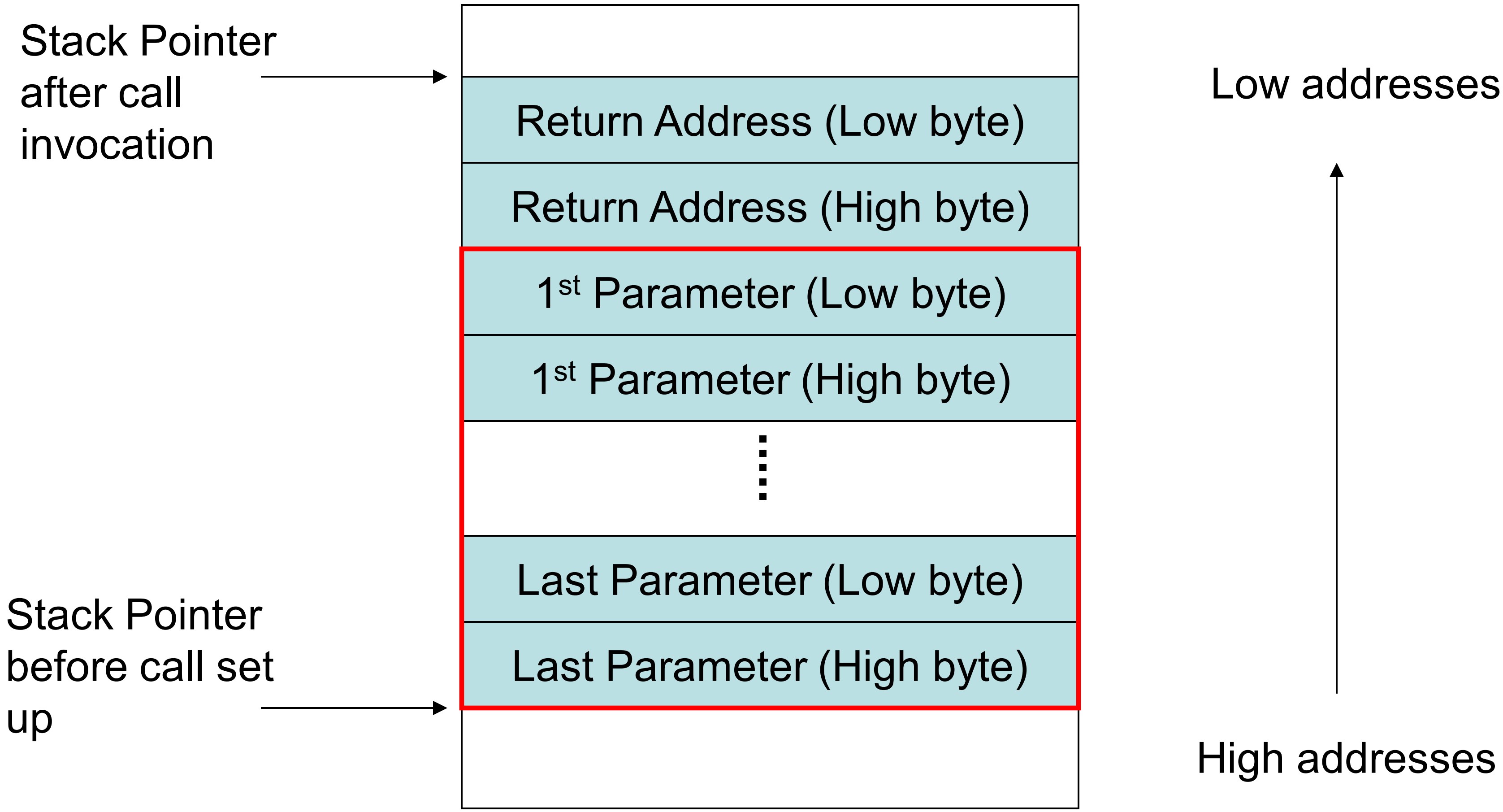
```
int result (int x, int y);

#asm root
result::
        // (sp+2) will contain first parameter x
        // 2 byte offset because of  the return address
        // stack grows to lower memory addresses with each push
        // (sp+4) will contain second parameter y

        ld hl,(sp+4)        ; get y
        ex de,hl            ; store in DE
        ld hl,(sp+2)        ; get x
        add hl, de          ; return result in HL

        ret
#endasm
```

Note: logical addresses for R4000 are 16 bit.



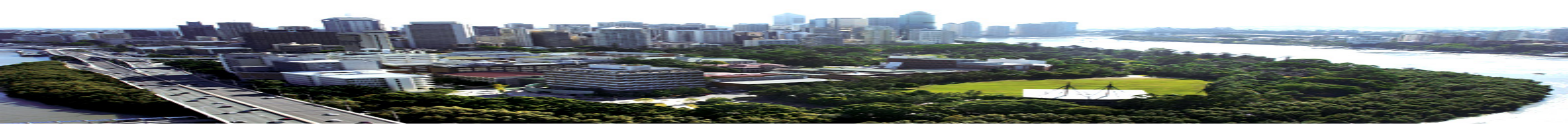


Passing parameters on the stack



Assembly implementation of C function with stack usage and pointers

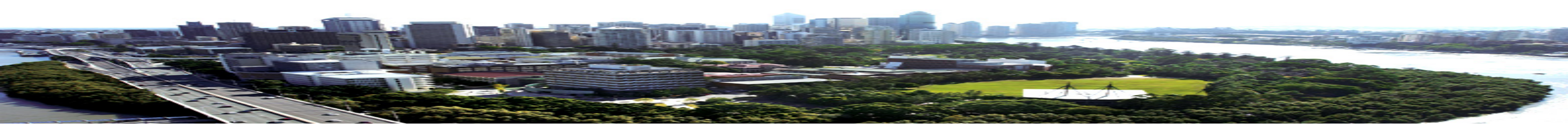
```
int testadd(int *px, int *py)              // note variable name and indexing in stack
{
 #asm
 ld hl, (sp + px)          // fetch pointer from the stack
 ld bcde, (hl)             // fetch parameter from memory using this
 ld bc, 0x0000             //  zero the extra bytes read
 ld hl, (sp + py)          // do the same for other parameter
 ld jkhl, (hl)
 ld jk, 0x0000
 add jkhl, bcde            // add the parameters, int result will be in hl
#endasm
}
```



Question

Write an assembly routine implementation of a Dynamic C function to multiply two unsigned integers and return an unsigned long.

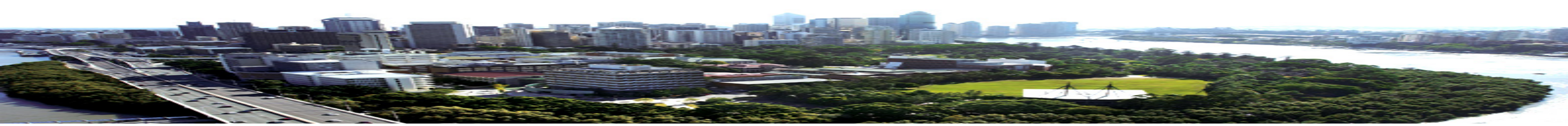
(Access the Dynamic C user manual)



Special symbols

Symbol	Description
`@SP`	Indicates the amount of stack space (in bytes) used for stack-based variables. This does not include arguments.
`@PC`	Constant for the current code location. For example: `ld hl, @PC` loads the code address of the instruction. ld hl,@PC+3 loads the address after the instruction since it is a 3 byte instruction.
`@RETVAL`	Evaluates the offset from the *frame reference point* to the stack space reserved for the `struct` function returns. See Section on page 158 for more information.
[redacted]	Determines the next reference address of a variable plus its [redacted]

*From the Dynamic C user manual



Example code for 64 bit addition

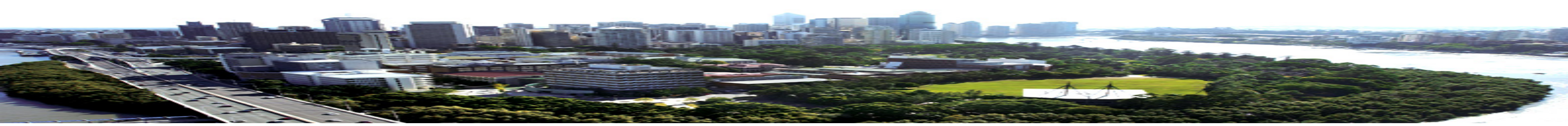
```
void eightadd( char *ch1, char *ch2 ){
#asm
ld hl,(sp+@SP+ch2) ;        get source pointer
ex de,hl ;                  save in register DE
ld hl,(sp+@SP+ch1) ;        get destination pointer
ld b,8 ;                    number of bytes
xor a ;                     clear carry
loop:
ld a,(de) ;                 ch2 source byte
adc a,(hl) ;                add ch1 byte
ld (hl),a ;                 store result to ch1 address
inc hl ;                    increment ch1 pointer
inc de ;                    increment ch2 pointer
djnz loop ;                 do 8 bytes
                            ;ch1 now points to 64 bit result
#endasm
}
```

Auto variables

– Temporary variables used by a function

– These are allocated storage on the *stack*

– A reentrant function uses auto variables; it should not have global variables.



Local variables on stack

	Stack
	⋮
SP-4 →	Auto variable 2 (int)
SP-2 →	Auto variable 1 (int)
SP →	Return address
	1st Parameter
	2nd Parameter
	⋮
	Last Parameter

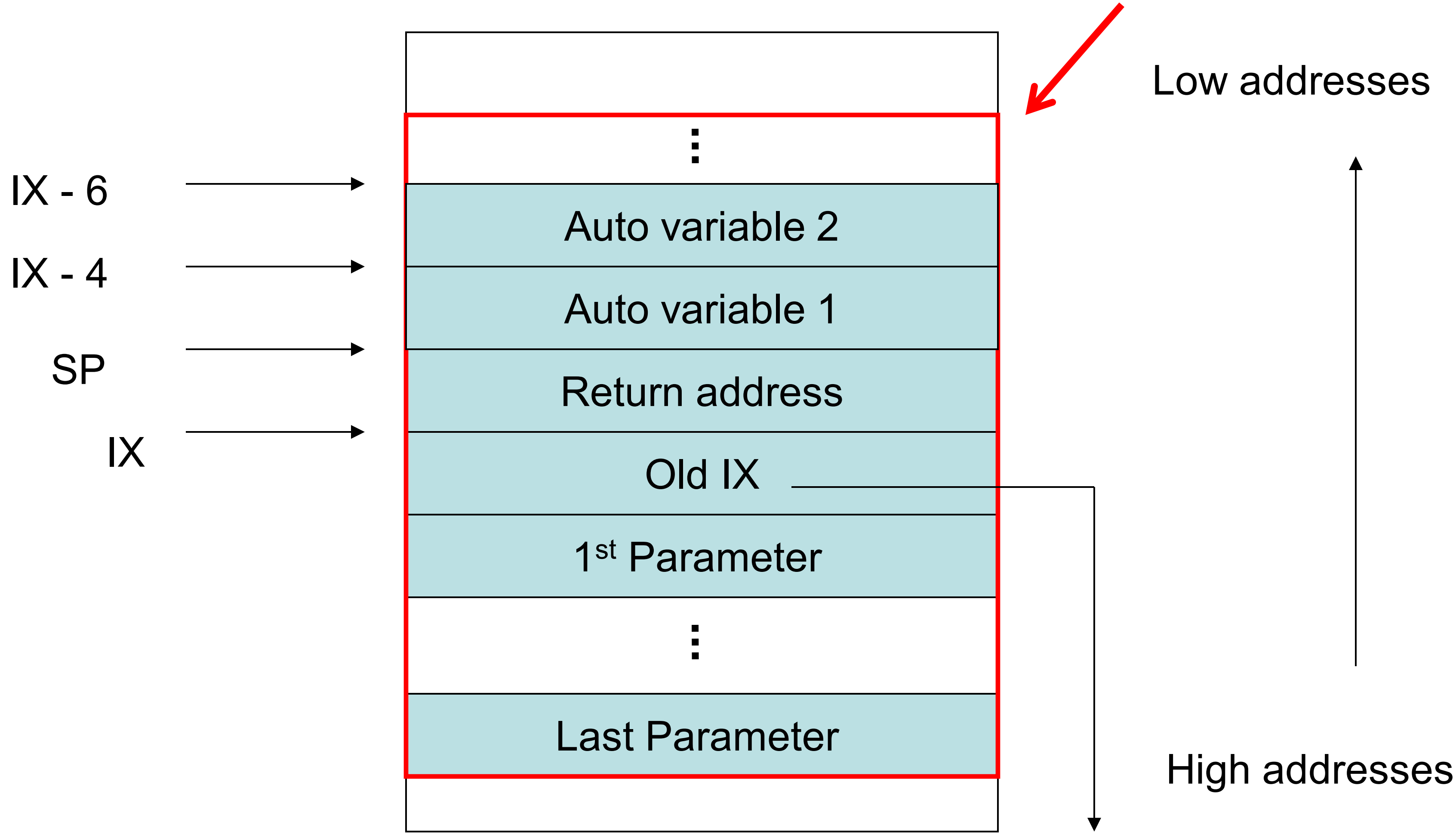
Low addresses

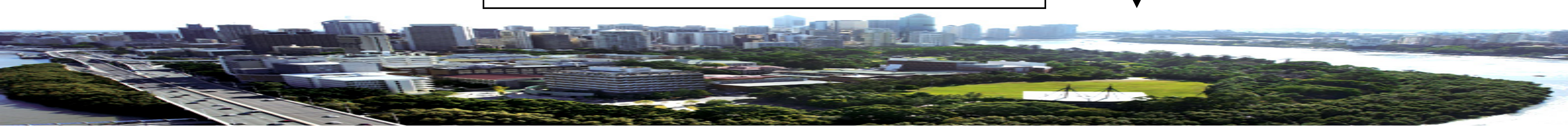
↑

High addresses

See Table 12.2.5 in the manual for special symbols like @SP which is the amount of stack space in bytes for auto variables

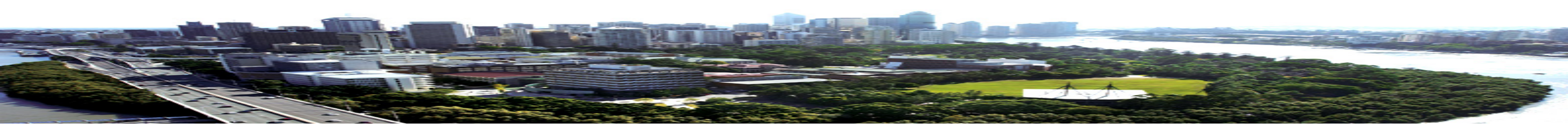
Local variable referencing using IX – stack frame





Library Functions for Port I/O

- Although assembly code may run faster, C functions are available in the libraries with Dynamic C for port I/O
- RdPortI(int PORT)
- BitRdPortI(into PORT, int bitcode)
- WrPortI(int PORT, char *PORTshadow, int value)
- BitWrPortI(int PORT, char *PORTshadow, int value, int bitcode)
- I stands for internal. Versions with suffix E for external.
- If NULL pointer is passed for the shadow port parameter, the shadow port is not updated.



Creating Time delays

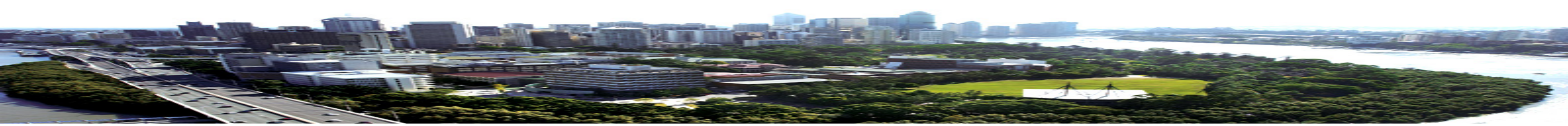
- Use assembly language or C statements to create delay loops (most precise interval if there are no interrupts or context switches)
- Set up timers and interrupts to create delays (processor can do other stuff, latencies may need to be considered)
- Use Dynamic C ‘s built-in delay functions such as DelaySec(10), DelayMs(50) . They use timed loops and the real time clock and work with cooperative multi-tasking. Dynamic C continuously updates system variables MS_TIMER, SEC_TIMER and TICK_TIMER. Latencies from context switches can vary.
- Use real-time kernel functions such as OSTimeDly(OS_TICKS_PER_SEC) . Latencies can be kept low.

Loop based delay

- Assembly better than high level language because clock cycles can be precisely counted and resulting delay is independent of the compiler.
- Rabbit at 59 MHz takes less than a microsecond for a 4 cycle instruction. Many instruction executions required even for a millisecond delay. A loop achieves this.

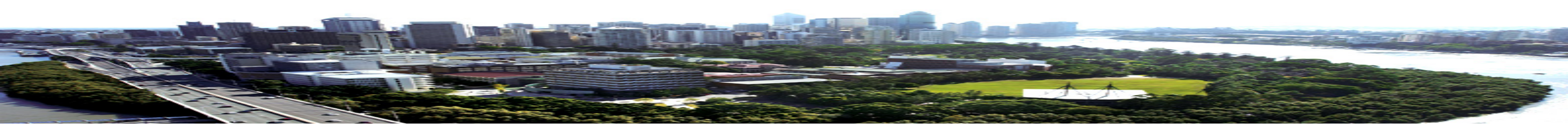
```
        LD A,0FFH       ;   4 cycles      loop counter
LOOP::  DEC A           ;   2 cycles
        JP   NZ, LOOP   ;   7 cycles
```

Need nested loops for larger delays.



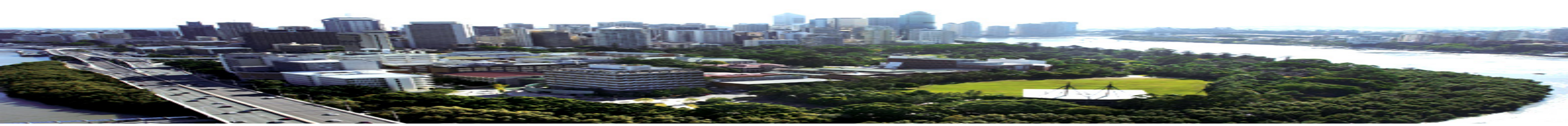
Assembly calling C – before the call (we don't need this)

- Save important registers prior to the call and restore them before return
- If the C function returns a struct, the calling program must reserve space on the stack for it
- If the assembly program needs to pass parameters to the C code, it has to push the parameters on the stack in the right order (last parameter pushed first)
- If the first argument is a pointer, an int, an unsigned int or char, it should be loaded into HL. If it is a long, unsigned long or float it should be loaded into the BC:DE combination.
- Finally, use CALL (note: return address is pushed by it)



Assembly calling C – after return
(we don't need this)

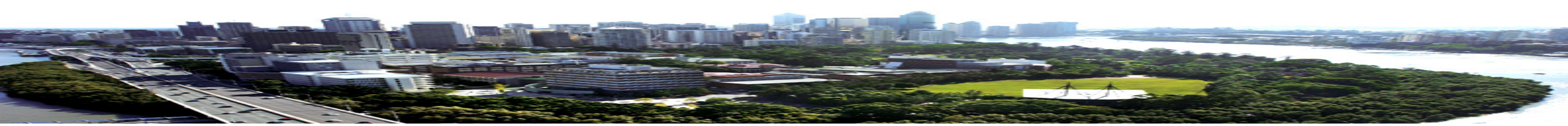
- Recover the stack space allocated to the arguments. Pop variables, 2 bytes at a time
- If the C program returns a struct, the calling program must recover the returned structure from the stack
- If the calling program saved any registers by pushing them on the stack, it should restore saved registers by popping them off the stack.
- The calling program should retrieve the returned value from HL (if an int, unsigned int or char) or BC:DE (if a long, unsigned long or float)



Demonstrations

Demonstration programs are placed on Blackboard.

Edit, Compile and Run
F4: Enter Edit Mode
F5: Compile code and load on target but don’t begin execution
F9: Compile code and load on target and begin execution
Debug
CTRL W: Add or delete a watch expression
CTRL U: Update watch expression window
F2: Set or remove a breakpoint
CTRL A: clear all breakpoints
F7: Single Step walking into functions
F8: Single step walking over functions
Highlight and CTRL H: provide help on a function



Conclusion

- Real-time embedded system programmers may need to interface assembly with C for speed, efficiency and low-level access.

- Dynamic C allows embedded code and stand-alone assembly. Library functions can be used for low level access when they are available.

- Stack operations and parameter passing/return must be understood when C and assembly are mixed.