Indirect Addressing Modes

Mode	Syntax	Effective Address and Operation		Notes
		Byte	Word	
RI	[*Wn*]	EA = (*Wn*)	EA = (*Wn*)	*Wn* unmodified
RI with pre-increment	[++*Wn*]	(*Wn* += 1); EA = (*Wn*)	(*Wn* + =2); EA = (*Wn*)	Increment before *Wn* used as EA
RI with pre-decrement	[--*Wn*]	(*Wn* −= 1); EA = (*Wn*)	(*Wn* −= 2); EA = (*Wn*)	Decrement before *Wn* used as EA
RI with post-increment	[*Wn*++]	EA = (*Wn*); (*Wn* += 1);	EA = (*Wn*); (*Wn* += 2);	Increment after *Wn* used as EA
RI with post-decrement	[*Wn*--]	EA = (*Wn*); (*Wn* −= 1);	EA = (*Wn*); (*Wn* −= 2);	Decrement after *Wn* used as EA
RI with register offset	[*Wn* + *Wb*]	EA = (*Wn*) + (*Wb*)	EA = (*Wn*) + (*Wb*)	*Wn*, *Wb* unmodified

Indirect Addressing Modes Examples

(a) Execute: mov [++W0], W2

W0 = 0x0802 + 2 = 0x0804 = EA;
(EA)→ W2 so (0x804) → W2

	Before	After	
W0	0x0802	0x0804	mod.
W1	0x0002	0x0002	
W2	0xFBC4	0xC7A6	mod.
0x0800	0x90B3	0x90B3	
0x0802	0x4E81	0x4E81	
0x0804	0xC7A6	0xC7A6	

(b) Execute: mov [W0++], W2

EA = W0 = 0x802 so (0x802) → W2;
W0 = 0x0802 + 2 = 0x0804

	Before	After	
W0	0x0802	0x0804	mod.
W1	0x0002	0x0002	
W2	0xFBC4	0x4E81	mod.
0x0800	0x90B3	0x90B3	
0x0802	0x4E81	0x4E81	
0x0804	0xC7A6	0xC7A6	

(c) Execute: mov.b [--W0], W2

W0 = 0x0802 – 1 = 0x0801 = EA;
(EA)→ W2 so (0x801) → W2

	Before	After	
W0	0x0802	0x0801	mod.
W1	0x0002	0x0002	
W2	0xFBC4	0xFB90	mod.
0x0800	0x90B3	0x90B3	
0x0802	0x4E81	0x4E81	
0x0804	0xC7A6	0xC7A6	

(d) Execute: mov.b [W0--], W2

EA = W0 = 0x802 so (0x802) → W2;
W0 = 0x0802 – 1 = 0x0801

	Before	After	
W0	0x0802	0x0801	mod.
W1	0x0002	0x0002	
W2	0xFBC4	0xFB81	mod.
0x0800	0x90B3	0x90B3	
0x0802	0x4E81	0x4E81	
0x0804	0xC7A6	0xC7A6	

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Register Indirect with Register Offset Examples

(a) Execute: mov [W0+W1], W2

EA = (W0)+(W1) = 0x0802+0x2 = 0x0804
(EA) → W2 so (0x804) → W2

	Before	After	
W0	0x0802	0x0802	
W1	0x0002	0x0002	
W2	0xFBC4	0xC7A6	mod.
0x0800	0x90B3	0x90B3	
0x0802	0x4E81	0x4E81	
0x0804	0xC7A6	0xC7A6	

(b) Execute: mov.b [W0+W1], W2

EA = (W0)+(W1) = 0x0802+0x2 = 0x0804
(EA) → W2 so (0x804) → W2

	Before	After	
W0	0x0802	0x0802	
W1	0x0002	0x0002	
W2	0xFBC4	0xFBA6	mod.
0x0800	0x90B3	0x90B3	
0x0802	0x4E81	0x4E81	
0x0804	0xC7A6	0xC7A6	

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Arrays and Pointers in C

- A pointer variable is a variable that contains the address of another variable.
- An array is a collection of like elements, such as an array of integers, array of characters, etc.
- One use of pointer variables in *C* is for stepping through the elements of an array.
- Another use of pointer variables is for passing arrays to subroutines
  - Only have to pass the address of the first element instead of passing all of the array elements!

A First Look at *C* Pointers

In *C*

```
uint16 u16_k, u16_j;
uint16* pu16_a;

(1)  u16_k = 0xA245;
(2)  u16_j = 0x9FC1;
(3)  pu16_a = &u16_j;
(4)  u16_k = *pu16_a;
```

(3) pu16_a contains address of u16_j

(4) *pu16_a is u16_j, so copy u16_j to k

In Memory

(a) Before (3)

Location	Contents	Variable
0x0800	0xA245	u16_k
0x0802	0x9FC1	u16_j
0x0804	0x????	pu16_a

(b) After (3)

Location	Contents	Variable	
0x0800	0xA245	u16_k	
0x0802	0x9FC1	u16_j	
0x0804	0x0802	pu16_a	mod.

(c) After (4)

Location	Contents	Variable	
0x0800	0x9FC1	u16_k	mod.
0x0802	0x9FC1	u16_j	
0x0804	0x0802	pu16_a	

*pu16_a in register transfer notation is ((pu16_a))
so *pu16_a $\rightarrow$ u16_k is ((pu16_a)) $\rightarrow$ u16_k, or (0x802) $\rightarrow$ u16_k

& is "address of" operator, "*" is dereference operator.
Pointers to data RAM are 16-bits wide because there are 64Ki addressable locations.

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

Same Example with uint32 variables

In *C*

```
uint32 u32_k, u32_j;
uint32* pu32_a;

(1)  u32_k = 0x76543210;
(2)  u32_j = 0xFEDCAB98;
(3)  pu32_a = &u32_j;
(4)  u32_k = *pu32_a;
```

(3) pu32_a contains address of u32_j

(4) *pu32_a is u32_j, so copy u32_ j to u32_k

In Memory

(a) Before (3)

Location	Contents	Variable
0x0800	0x3210	u32_k.LSW
0x0802	0x7654	u32_k.MSW
0x0804	0xBA98	u32_j.LSW
0x0806	0xFEDC	u32_j.MSW
0x0808	0x????	pu32_a

(b) After (3)

Location	Contents	Variable	
0x0800	0x3210	u32_k.LSW	
0x0802	0x7654	u32_k.MSW	
0x0804	0xBA98	u32_j.LSW	
0x0806	0xFEDC	u32_j.MSW	
0x0808	0x0804	pu32_a	mod.

(c) After (4)

Location	Contents	Variable	
0x0800	0xBA98	u32_k.LSW	mod.
0x0802	0xFEDC	u32_k.MSW	mod.
0x0804	0xBA98	u32_j.LSW	
0x0806	0xFEDC	u32_j.MSW	
0x0808	0x0804	pu32_a	

p is still 16-bits! Pointer size is always 16 bits, not dependent upon referenced data size

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

uint16* Pointer Example to Assembly

In *C*

uint16 u16_k, u16_j;
uint16* pu16_a;

(1) u16_k = 0xA245;

(2) u16_j = 0x9FC1;

(3) pu16_a = &u16_j;

(4) u16_k = *pu16_a;

In Assembly

```
u16_k: .space 2
u16_j: .space 2
 ;; W1 is used for pu16_a

mov #0xA245,W0
mov W0,u16_k              ;u16_k = 0xA245

mov #0x9FC1,W0
mov W0,u16_j              ;u16_j = 0x9FC1

mov #u16_j,W1             ;W1 = &u16_j

mov [W1],W0               ;W0 = *W1 = *pu16_a = u16_j
mov W0,u16_k              ;u16_k = W0 = u16_j
```

W1 register used to implement the pu16_a pointer.
Indirect addressing used to implement *pu16_a.

Pointers in assembly

Operation	RTL	Assembly	C

Literal	____
	0x0900 → W0
		#
		mov #0x0900, W0
		mov #u8_c, W0
		mov W0, pu8_b
		mov #au8_d, W0
		mov W0, pu8_b
			____ (based on type)
			W0 = 0x0900;
			pu8_b = &u8_c;
			=> pu8_b = 0x0804;
			pu8_b = au8_d;
			=> pu8_b = 0x0805;

Direct	(___) (u16_a) → W0 => (0x0800) → W0	____ mov u16_a, W0 => mov 0x0800, W0	____ (based on type) W0 = u16_a;

(Register) indirect	((___))
	((pu8_b)) → W0
	((0x0802)) → W0
		[___] (**registers only**)
		mov pu8_b, W1
		mov [W1], W0
			*____ or ____[0]
			W0 = *pu8_b; // or
			W0 = pu8_b[0]

```
.space u16_a   2  ; u16_a = 0x0800
.space pu8_b   2  ; pu8_b = 0x0802
.space u8_c    1  ; u8_c = 0x0804
.space au8_d  10  ; au8_d = 0x0805
```

Address	Data
0x0800	0x1234
0x0802	0x0804
0x0804	0x5678

uint32* Pointer Example to Assembly

In *C*

```
uint32 u32_k, u32_j;
uint32* pu32_a;

(1)  u32_k = 0x76543210;
(2)  u32_j = 0xFEDCAB98;

(3)  pu32_a = &u32_j;

(4)  u32_k = *pu32_a;
```

In Assembly

```
u32_k: .space 4   ;W1 is used for pu32_a
u32_j: .space 4
```

(1), (2):

```
...32-bit initialization of u32_k not shown
...32-bit initialization of u32_j not shown
```

(3):

```
mov #u32_j,W1        ;W1=&u32_j
```

(4a) 5 instr. cycles, 3 instr. words

```
mov.d [W1],W2        ;W2=u32_j.LSW, W3=u32_j.MSW
mov #u32_k,W1        ;W1=&u32_k
mov.d W2,[W1]        ;*W1=W3:W2=u32_j.MSW:u32_j.LSW
```

(4b) 5 instr. cycles, 5 instr. words

```
mov [W1],W0          ;W0= u32_j.LSW
mov W0,u32_k         ;u32_k.LSW = W0 = u32_j.LSW
inc2 W1,W1           ;W1 is now address of u32_j.MSW
mov [W1],W0          ;W0= u32_j.MSW
mov W0,u32_k+2       ;u32_k.MSW = W0 = u32_j.MSW
```

(4c) 4 instr. cycles, 4 instr. words

```
mov [W1++],W0        ;W0= u32_j.LSW, W1+=2
mov W0,u32_k         ;u32_k.LSW = u32_j.LSW
mov [W1],W0          ;W0= u32_j.MSW
mov W0,u32_k+2       ;u32_k.MSW = u32_j.MSW
```

(4d) 3 instr. cycles, 3 instr. words

```
mov #u32_k, W2       ;W2 = &u32_k
mov [W1++],[W2++]    ;u32_k.LSW = u32_j.LSW,
                     ;W1+=2,W2+=2
mov [W1],[W2]        ;u32_k.MSW = u32_j.MSW
```

Pointer Arithmetic

Pointer arithmetic means to add or subtract a pointer by some value. The value being added to the pointer or subtracted from the pointer is multiplied by the SIZE in bytes of what the pointer is pointing at!

```
uint8*   pu8_a;
uint8    u8_j, u8_k;

 pu8_a = &u8_j;
 pu8_a = pu8_a + 1;       ; pu8_a = pu8_a + 1*sizeof(uint8)
                          ; pu8_a = pu8_a + 1*1 = pu8_a + 1;
```

```
uint16*  pu16_a;
uint16   u16_j,u16_k;

 pu16_a = &u16_j;
 pu16_a = pu16_a + 1;     ; pu16_a = pu16_a + 1*sizeof(uint16)
                          ; pu16_a = pu16_a + 1*2 = pu16_a + 2;
```

```
uint32*  pu32_a;
uint32   u32_j,u32_k;

 pu32_a = &u32_j;
 pu32_a = pu32_a + 1;     ; pu32_a = pu32_a + 1*sizeof(uint32)
                          ; pu32_a = pu32_a + 1*4 = pu32_a + 4;
```

Pointer Arithmetic (continued)

Pointer arithmetic means to add or subtract a pointer by some value. The value being added to the pointer or subtracted from the pointer is multiplied by the SIZE in bytes of what the pointer is pointing at!

`uint8* pu8_a;`
`uint8 u8_j, u8_k;`
 	
`pu8_a = &u8_j;`
`pu8_a = pu_a + 1;`
	`mov #u8_j, W1 ;use W1 for pu8_a`
	`add W1,#1, W1 ;pu8_a = pu8_a + 1*sizeof(uint8)`
	`;pu8_a = pu8_a + 1*1 = pu8_a + 1;`

`uint16* pu16_a;`
`uint16 u16_j,u16_k;`
 	
`pu16_a = &u16_j;`
`pu16_a = pu_a + 1;`
	`mov #u16_j, W1 ;use W1 for pu16_a`
	`add W1,#2, W1 ;pu16_a = pu16_a + 1*sizeof(uint16)`
	`;pu16_a = pu16_a + 1*2 = pu16_a + 2;`

`uint32* pu32_a;`
`uint32 u32_j,u32_k;`
 	
`pu32_a = &u32_j;`
`pu32_a = pu_a + 1;`
	`mov #u32_j, W1 ;use W1 for pu32_a`
	`add W1,#4, W1 ;pu32_a = pu32_a + 1*sizeof(uint32)`
	`;pu32_a = pu32_a + 1*4 = pu32_a + 4;`

Arrays and Pointers: Array of uint8

In *C*

uint8 au8_x[4] = {0x05,
    0xAB, 0x72, 0x36};
uint8 *pu8_y;

(1) au8_x[2] = au8_x[1];

(2) pu8_y = &au8_x[2];

(3) pu8_y++;

`pu8_y++` is
`pu8_y+1*sizeof(uint8)` is
`pu8_y+1*1` is
`pu8_y+1`

In Memory

	Location	Contents	Variable	
Before (1)	0x0800	0xAB 05	au8_x[1],au8_x[0]	
	0x0802	0x36 72	au8_x[3],au8_x[2]	
	0x0804	0x????	pu8_y	

au8_x[1] copied to au8_x[2]

	Location	Contents	Variable	
After (1)	0x0800	0xAB 05	au8_x[1],au8_x[0]	
	0x0802	0x36 AB	au8_x[3],au8_x[2]	mod.
	0x0804	0x????	pu8_y	
After (2)	0x0800	0xAB 05	au8_x[1],au8_x[0]	
	0x0802	0x36 AB	au8_x[3],au8_x[2]	
	0x0804	0x0802	pu8_y	mod.

+1

	Location	Contents	Variable	
After (3)	0x0800	0xAB 05	au8_x[1],au8_x[0]	
	0x0802	0x36 AB	au8_x[3],au8_x[2]	
	0x0804	0x0803	pu8_y	mod.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Arrays and Pointers: Array of uint16

In *C*

```
uint16 au16_x[4] = {0x38A0,
                    0xC9F5,
                    0xB861,
                    0x724D};
uint16 *pu16_y;

  (1)  au16_x[2] = au16_x[1];

  (2)  pu16_y = &au16_x[2];

  (3)  pu16_y++;
```

```
pu16_y++                       is
pu16_y+(1*sizeof(uint16))      is
pu16_y+(1*2)                   is
pu16_y+2
```

In Memory

	Location	Contents	Variable	
Before (1)	0x0800	0x38A0	au16_x[0]	
	0x0802	0xC9F5	au16_x[1]	
	0x0804	0xB861	au16_x[2]	
	0x0806	0x724D	au16_x[3]	
	0x0808	0x????	pu16_y	
After (1)	0x0800	0x38A0	au16_x[0]	
	0x0802	0xC9F5	au16_x[1]	
	0x0804	0xC9F5	au16_x[2]	mod.
	0x0806	0x724D	au16_x[3]	
	0x0808	0x????	pu16_y	
After (2)	0x0800	0x38A0	au16_x[0]	
	0x0802	0xC9F5	au16_x[1]	
	0x0804	0xC9F5	au16_x[2]	
	0x0806	0x724D	au16_x[3]	
	0x0808	0x0804	pu16_y	mod.
After (3)	0x0800	0x38A0	au16_x[0]	
	0x0802	0xC9F5	au16_x[1]	
	0x0804	0xC9F5	au16_x[2]	
	0x0806	0x724D	au16_x[3]	
	0x0808	0x0806	pu16_y	mod.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Add two uint16 Arrays

In *C*

```
uint16 au16_a[10];
uint16 au16_b[10];
uint16 au16_c[10];
uint8 u8_i;

for (u8_i = 0; u8_i < 10; u8_i++){

 au16_c[u8_i] = au16_a[u8_i] +
                au16_b[u8_i];

}
```

In Assembly

```
au16_a  .space  10*2  ;each array occupies
au16_b  .space  10*2  ;20 bytes since each
au16_c  .space  10*2  ;element is 2 bytes
 ;W1 is used to point at au16_a
 ;W2 is used to point at au16_b
 ;W3 is used to point at au16_c
 ;W4 is loop counter (u8_i)

 mov #au16_a,W1        ;W1 = &au16_a[0]
 mov #au16_b,W2        ;W2 = &au16_b[0]
 mov #au16_c,W3        ;W3 = &au16_c[0]
 clr.b W4              ;clear loop counter
top_loop:
 cp.b W4,#10           ;check loop counter
 bra GEU,end_loop      ;exit if finished
 mov [W1++],W0         ;W0 = *W1, W1++
 add W0,[W2++],[W3++]  ;*W3 = *W2 + W0
                       ;W3++,W2_ptr++
 inc.b W4,W4       ;increment loop counter
 bra top_loop      ;loop back to top
end_loop:
  ...rest of code...
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

C Strings

(a) In *C*

```
char sz_a[] = "Hello";
char* psz_x;

psz_x = &sz_a[0];
while (*psz_x != 0) {
  //convert to upper case
  if (*psz_x > 0x60 &&
      *psz_x < 0x7B)  {
   //lowercase 'a' - 'z', so
   //convert to 'A' - 'Z'
   *psz_x = *psz_x - 0x20;
   }
   psz_x++;  //advance to
        //next character
  }
```

(b) In Memory

Location	Contents	Variable	as ASCII
0x0800	0x6548	sz_a[1],sz_a[0]	'e', 'H'
0x0802	0x6C6C	sz_a[3],sz_a[2]	'l','l'
0x0804	0x006F	sz_a[5],sz_a[4]	null, 'o'

(c) In Assembly

```
  ;W0 is used to implement psz_x
  ;W1 is used to hold contents of *psz_x
  ;W2 is used a temp. reg to hold constants

  mov #sz_a,W0       ;W0 = &sz_a[0]
top_loop:
  mov.b [W0],W1      ;W1 = *psz_x
  cp.b W1,#0x00
  bra Z, end_loop    ;exit if at end of string
  mov #0x60,W2
  cp.b W1,W2         ;compare *psz_x and 0x60
  bra LEU, end_if    ;skip if-body
  mov #0x7B,W2
  cp.b W1,W2         ;compare *psz_x and 0x7B
  bra GEU, end_if    ;skip if_body
  mov #0x20,W2
  sub.b W1,W2,[W0]   ;*psz_x = *psz_x-0x20
end_if:
  inc W0,W0          ;psz_x++
  bra top_loop       ;loop back to top
end_loop:
  ...rest of code...
```


From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

The repeat Instruction

(a) In *C*

```
uint16  au16_x[64];
uint8  u8_i;

// Initialize contents of au16_x[]
// to zero
for (u8_i = 0; u8_i < 64; u8_i++) {
  au16_x[u8_i] = 0;
}
```

(b) In Assembly

```
  ;W1 is used to point at au16_x
  ;W2 is used as loop counter
  mov  #au16_x,W1    ;W1 points at &au16_x[0]
  clr.b W2           ;clear loop counter
  mov.b #64,W3       ;W3 holds loop max count
top_loop:
  cp.b W2,W3         ;check loop counter
  bra GEU,end_loop   ;exit if finished
  clr [W1++]         ;au16_x[u8_i] = 0;
  inc.b W2,W2        ;increment loop counter
  bra top_loop       ;loop back to top
end_loop:
  ...rest of code...
----------------------------------------------
```

(c) In Assembly (use the *repeat* instruction)

```
  ;W1 is used to point at au16_x
  mov  #au16_x,W1    ;W1 points at &au16_x[0]
  repeat #63         ;repeat next instruction!
  clr [W1++]         ;au16_x[u8_i] = 0;
  ...rest of code...
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Why are Subroutines needed?

Without Subroutines

```
instr_1
instr_2
instr_a1
instr_a2
...
instr_an
instr_3
instr_4
instr_5
instr_a1
instr_a2
...
instr_an
instr_6
instr_7
...
```

Replicated instruction sequence

With Subroutines

Caller

```
instr_1
instr_2
call sub
instr_3
instr_4
instr_5
call sub
instr_a2
...
```

Callee

```
instr_a1
instr_a2
...
instr_an
return
```

Replicated instruction sequence as a subroutine

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

A *C* Subroutine (*Function*)

General form of a *C* subroutine is:

(*return_type*) *subname* (*parm list*)
{
  local_variable_decl;
  subroutine_body;
  return *return_value*;
}

`countOnes` Subroutine

```
// count "1" bits in uint16 parameter
uint8 countOnes (uint16 u16_v) {
  uint8 u8_cnt, u8_i;

  u8_cnt = 0;
  for (u8_i = 0; u8_i < 16; u8_i++) {
    if (u16_v & 0x0001) u8_cnt++;
    u16_v = u16_v >> 1;
  }
  return u8_cnt;
}

main (void) {
  uint16 u16_k;
  uint8  u8_j;

  u16_k = 0xA501;
  u8_j = countOnes(u16_k);
  printf (
  "Number of one bits in %x is %d\n",
  u16_k, u8_j);
}
```

parameter list: gives types and names

subroutine body

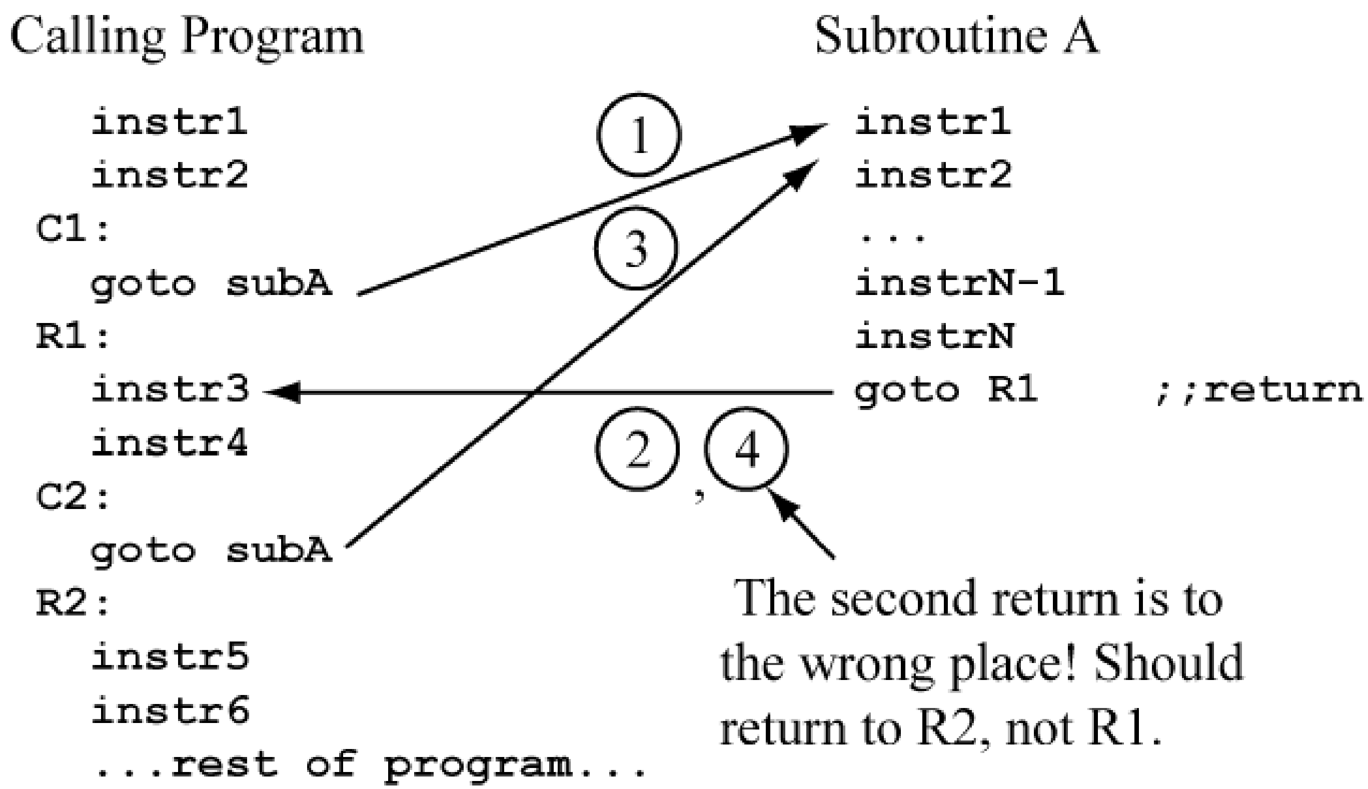
subroutine return

main program

subroutine call

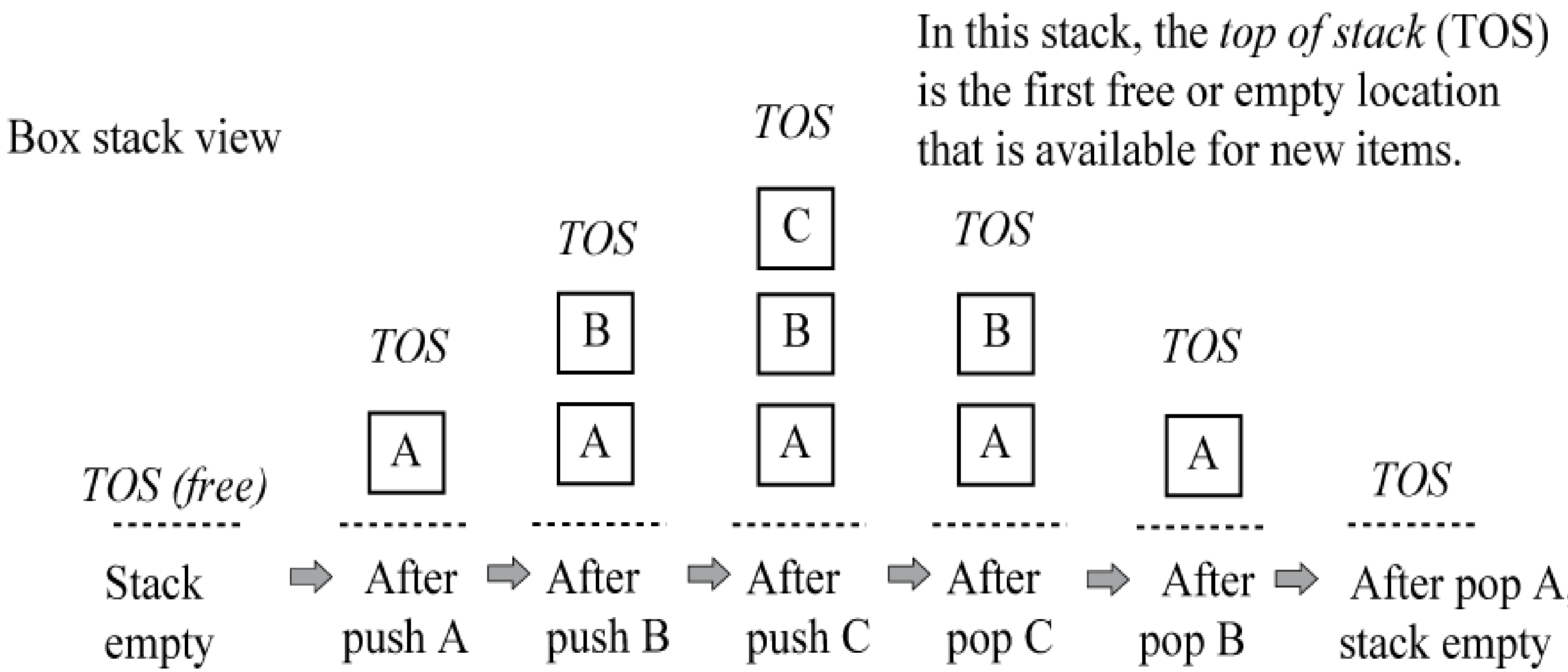
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Call/Return



From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

A Stack



From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Push/Pop on PIC24 Stack

(a) *push* Operation: *dataA* → (SP), SP = SP + 2

mov *src*, [W15++]

W15 is the *stack pointer (*SP), *dataA* is specified by the source addressing mode.

Stack grows towards increasing memory locations

Before push of *dataA*

Location	Contents	
0x0900	????	← SP
0x0902	????	
0x0904	????	
W15	0x0900	

After push of *dataA*

Location	Contents	
0x0900	*dataA*	modified
0x0902	????	← SP
0x0904	????	
W15	0x0902	modified

(b) *pop* Operation: SP = SP – 2, ((SP)) → destination as specified by destination addressing mode

mov [--W15], *dest*

Before pop operation

Location	Contents	
0x0900	*dataA*	
0x0902	????	← SP
0x0904	????	
W15	0x0902	

After pop operation

Location	Contents	
0x0900	*dataA*	← SP
0x0902	????	
0x0904	????	
W15	0x0900	modified

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Push/Pop Forms

Name	Mnemonic	Operation
Push	`push Wso` `push f`	`(Wso) → (W15); (W15)+2 → W15` `(f) → (W15); (W15)+2 → W15`
Push double word	`push.d Wns`	`(Wns) → (W15);(W15)+2 → W15` `(Wns+1) → (W15);(W15)+2 → W15`
Pop	`pop Wdo` `pop f`	`(W15)-2 → W15; (W15) → Wdo` `(W15)-2 → W15; (W15) → f`
Pop double word	`pop.d Wnd`	`(W15)-2 → W15; (W15) → Wnd` `(W15)-2 → W15; (W15) → Wnd+1`

f specifies a word address anywhere in the lower 32 Ki words of data memory

Field	Description
Wnd	One of 16 destination working registers ∈ {W0..W15}
Wns	One of 16 source working registers ∈ {W0..W15}
Ws	Source W register ∈ { Ws, [Ws], [Ws++], [Ws--], [++Ws], [--Ws] }
Wso	Source W register ∈ { Wns, [Wns], [Wns++], [Wns--], [++Wns], [--Wns], [Wns+Wb] }
Wnd	One of 16 destination working registers ∈ {W0..W15}
Wdo	Destination W register ∈ { Wnd, [Wnd], [Wnd++], [Wnd--], [++Wnd], [--Wnd],[Wnd+Wb] }
Wn	One of 16 working registers ∈ {W0..W15}

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Push/Pop Example

(a) Execute: `push W0` (W0) → (SP); SP = SP + 2

Equivalent to: `mov W0,[W15++]`

Before

Location	Contents	
0x0800	0x9ED1	
0x0802	0x60C8	
0x0950	0xA23B	← SP
0x0952	0x3F90	
W0	0xCD18	
W1	0x0802	
W15	0x0950	

After

Location	Contents	
0x0800	0x9ED1	
0x0802	0x60C8	
0x0950	0xCD18	modified
0x0952	0x3F90	← SP
W0	0xCD18	
W1	0x0802	
W15	0x0952	modified, SP = SP + 2

(b) Execute: `pop 0x0800` SP = SP – 2; ((SP)) → 0x0800

Semantically equivalent to: `mov [--W15],0x0800`

However, this addressing mode combination is illegal for the `mov` instruction.

Before

Location	Contents	
0x0800	0x9ED1	
0x0802	0x60C8	
0x0950	0xCD18	
0x0952	0x3F90	← SP
W0	0xCD18	
W1	0x0802	
W15	0x0952	

After

Location	Contents	
0x0800	0xCD18	modified
0x0802	0x60C8	
0x0950	0xCD18	← SP
0x0952	0x3F90	
W0	0xCD18	
W1	0x0802	
W15	0x0950	modified, SP = SP – 2

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

New instructions: `call/rcall`, `return`

The `call/rcall` instructions can used to call a subroutine. The `call` instruction is 2 program words, the `rcall` is 1 program word. Each does the same two things:

1. Pushes the *return address* on the stack. The return address is the address of the instruction after the `call/rcall` instruction.
2. Does an unconditional jump to the subroutine.

The `return` instruction is used to return from a subroutine. It pops the top of the stack into the program counter, causing a jump to that location. Under normal conditions, the value popped from the stack will be the return address for the `call/rcall` instruction used to jump to this subroutine.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Call/Return and the Stack

```
Location  Contents
                main
                  instr?
0x0240 020268     call  sub_a    (1)
0x0242 000000     nop
0x0244 ??????     instr?
                here:
                  goto here

                sub_a:
0x0268 ??????     instr?
0x026A 02033A     call  sub_b    (2)
0x026C 000000     nop
0x026E ??????     instr?
                  return         (6)
                sub_b:
0x033A ??????     instr?
0x033C 0700E1     rcall  sub_c   (3)
0x033E ??????     instr?
                  return         (5)
                sub_c:
0x0500 ??????     instr?
                  instr?
                  return         (4)
```

	Location	Contents	
Initial SP →	0x0900	0x0244	(6) pop
push 0x000244 (1)	0x0902	0x0000	
	0x0904	0x026E	(5) pop
push 0x00026E (2)	0x0906	0x0000	
	0x0908	0x033E	(4) pop
push 0x00033E (3)	0x090A	0x0000	
	0x090C	0x????	

For `call`, return address = PC + 4
For `rcall`, return address = PC + 2

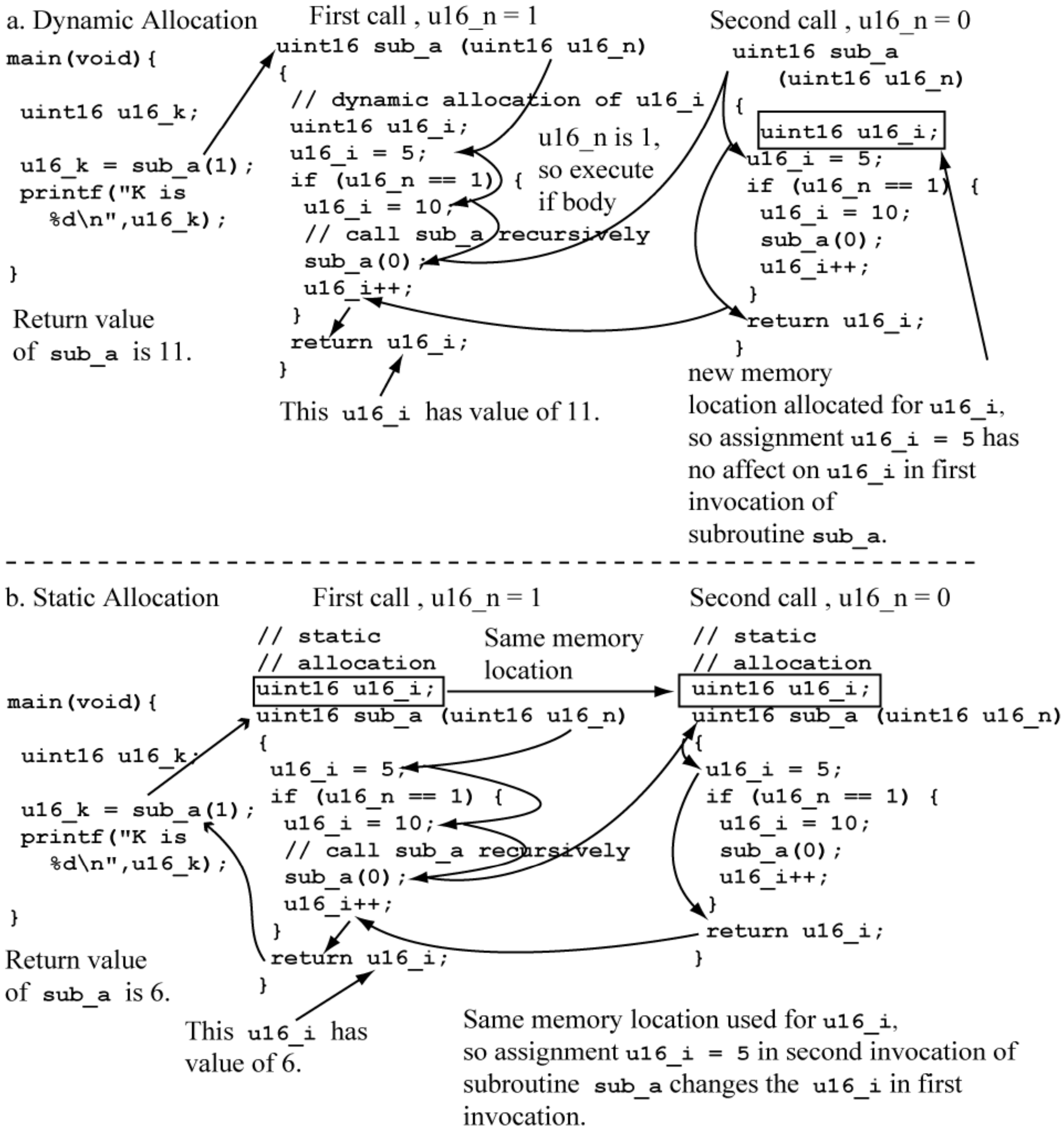
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Call/Return Forms

Name	Mnemonic	Operation
Call	`call` *`label_lit23`*	Push return address on stack, new SP = SP + 4, then *label_lit23* → PC
	`call` *`Wn`*	Push return address on stack, new SP = SP + 4, then (*Wn*) → PC
Relative call	`rcall` *`label_slit16`*	Push return address on stack, new SP = SP + 4, then (PC) + (2**label_slit16*) → PC
	`rcall` *`Wn`*	Push return address on stack, new SP = SP + 4, then (PC) + (2**Wn*) → PC
Return	`return`	Pop return address from stack into the PC, new SP = SP - 4
Return with literal in *Wn*	`retlw{.b}` *`#lit10,Wn`*	Pop return address from stack into the PC, new SP = SP - 4,and #lit10 → *Wn*

The `call` *`label_lit23`* instruction is 2 instruction words, all others are 1 instruction word.

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".



Dynamic Allocation for Locals

Dynamic allocation is needed for recursive functions to operate correctly.

New space for parameters and locals are allocated in registers or on the stack.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Rules For Subroutine Parameter Passing

• W0-W7 are used for parameters, left to right order. W0-W7 are *caller* saved (if caller wants these preserved, caller has to save them).
• Function values returned in W0-W3 (W0 for 8/16 bit, W0-W1 for 32-bit, W0-W3 for 64-bit values).
• Registers W8-W14 are *callee* saved (if the callee uses them, must be preserved).
• Locals are allocated to unused W0-W7 registers, and also to W8-W14.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Subroutine Example

In *C*

```c
// count "1" bits in uint16 parameter
uint8 countOnes (uint16 u16_v) {
  uint8 u8_cnt, u8_i;

  u8_cnt = 0;
  for (u8_i = 0; u8_i < 16; u8_i++) {
    if (u16_v & 0x0001) u8_cnt++;
    u16_v = u16_v >> 1;
  }
  return u8_cnt;
}
```

In Assembly

```
; u16_v passed in W0
; return value passed back in W0
; W1 used for local u8_cnt, W2 for u8_i
countOnes:
    clr.b W1            ;u8_cnt=0
    clr.b W2            ;u8_i=0
loop_top:
    cp.b W2,#16         ;compare u8_i, 16
    bra GEU,end_loop    ;exit loop if u8_i>=16
    btst.z W0,#0        ;test LSbit for zero
    bra Z, end_if
    inc.b W1,W1         ;u8_cnt++;
end_if:
    lsr W0,#1,W0        ;u16_v = u16_v >> 1
    inc.b W2,W2         ;u8_i++
    bra loop_top
end_loop:
    mov.b W1,W0         ;W0 = u8_cnt for
    return              ;  return value
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Subroutine Call Example

In *C*

```
uint16 u16_k;
uint8  u8_j;

main (void) {

u16_k = 0xA501;

u8_j = countOnes(u16_k);

}
```

In Assembly

```
u16_k:    .space 2
u8_j:     .space 1

.text
__reset:                        ;reset entry point
  mov #__SP_init, W15           ;init SP
  mov #__SPLIM_init,W0
  mov W0, SPLIM                 ;init SPLIM
  rcall   main                  ;call main()
  reset                         ;start over

main:
  mov #0xA501, W0
  mov WREG,u16_k           ;init u16_k

 ;subroutine call
  mov u16_k,WREG           ;W0 used for
  rcall countOnes          ;u16_v parameter
  mov.b WREG,u8_j            ;return value in W0
done:
  bra done                 ;do not return
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Using the stack to save register values

Recall the `countOnes()` PIC24 implementation used the W1, W2 registers for local variables. What if the caller wanted to save these registers? Push them on the stack before call, then pop them off (must be in reverse order of push!!!)

```
push W1               ;save W1
push W2               ;save W2
mov u16_k, WREG       ;pass u16_k in W0
rcall countOnes       ;call the function
mov.b WREG, u8_j      ;save return value
pop W2                ;restore W2
pop W1                ;restore W1
```

A Recursive Subroutine

In *C*

```
 uint16 fib  (uint16 u16_n) {
 uint16 u16_f1;

 if (u16_n == 0) return 0; //(1)
 if (u16_n == 1) return 1; //(2)
 u16_f1 = fib(u16_n-1);   //(3)
 u16_f1 = fib(u16_n-2)
          + u16_f1;        //(4)
 return (u16_f1);
}
/* Fib sequence:
0,1,1,2,3,5,8,13,21,34,55,89,
144,233,377,610,987,...*/
main (void) {
 uint16  u16_k, u16_j;
 u16_j = 13;
 u16_k =fib(u16_j);
 printf ("fib(%d) is: %d\n",
    u16_j,u16_k);
}
```

In Assembly

```
;only fib subroutine is shown
;u16_n passed in W0
;return value passed back in W0
;W1 used for f1 local
fib:
    cp W0,#0              ;u16_n == 0?
    bra Z,fib_exit        ;if W0 is 0, return it
    cp W0,#1              ;u16_n == 1?
    bra Z,fib_exit        ;if W0 is 1, return it

    push W0               ;save u16_n (W0)
    dec W0,W0             ;u16_n = u16_n-1
    rcall fib             ;fib(u16_n-1)
    mov W0,W1             ;save u16_f1 in W1
    pop W0                ;restore old u16_n

    dec2 W0,W0            ;u16_n = u16_n-2
    push W1               ;save u16_f1
    rcall fib             ;fib(u16_n-2)
    pop W1                ;restore u16_f1 to W1
    add W0,W1,W0          ;W0 = fib(u16_n-2)+u16_f1
fib_exit:
    return                ;return u16_f1
```

Observe that stack is used for temporary storage.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Subroutine with Multiple Parameters

In *C*

```
void swapU32  (
   uint32* pu32_b,
   uint8 u8_k, uint8 u8_j) {
 uint32 u32_t;

 u32_t = pu32_b[u8_k]; //1
 pu32_b[u8_k] =
              pu32_b[u8_j]; //2
 pu32_b[u8_j] = u32_t;  //3
//NOTE: an access such as
// pu32_b[u8_k]  is the same as
//*(pu32_b+u8_k)
}

uint32 au32_x[]={0x0982A1F9,
   0x88B23204, 0xE3D96B10,
   0xC385FB29};

main (void) {
 printf ("::%lx, %lx, %lx, %lx\n",
          au32_x[0],au32_x[1],
          au32_x[2],au32_x[3]);
 swapU32(au32_x,1,3);       //4
 printf ("::%lx, %lx, %lx, %lx\n",
          au32_x[0],au32_x[1],
          au32_x[2],au32_x[3]);
}
```

In Assembly

```
; pu32_b passed in W0
; u8_k passed in W1
; u8_j passed in W2
; W3 used for (pu32_b+k)
; W4 used for (pu32_b+j)
; W5,W6 used for u32_t
swapU32:
  sl W1,#2,W1          ;u8_k=u8_k<<2=u8_k*4
  sl W2,#2,W2          ;u8_j=u8_j<<2=u8_j*4
  add W0,W1,W3         ;W3 = pu32_b+u8_k
  add W0,W2,W4         ;W4 = pu32_b+u8_j

  mov [W3++],W5        ;u32_t=pu32_b[u8_k] LSW
  mov [W3--],W6        ;u32_t=pu32_b[u8_k] MSW

  mov [W4++],[W3++]    ;pu32_b[u8_k] LSW
                       ; = pu32_b[u8_j] LSW
  mov [W4--],[W3--]    ;pu32_b[u8_k] MSW
                       ; = pu32_b[u8_j] MSW

  mov W5,[W4++]        ;pu32_b[u8_j]=u32_t LSW
  mov W6,[W4--]        ;pu32_b[u8_j]=u32_t MSW

  return

main:                  ;init parms
  mov #au32_x,W0       ;W0 = &au32_x[0]
  mov #1,W1            ;W1 = 1
  mov #3,W2            ;W2 = 2
  rcall swapU32        ;call the subroutine
done:
   goto    done        ;infinite wait loop
```

The operation `pu32_b[u8_k]` is the same as `*(pu32_b+u8_k)`

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Global Variable Initialization

In *C*

```
void upcase (char* psz_x){
while (*psz_x != 0) {
  //convert to upper case
  if (*psz_x > 0x60 &&
       *psz_x < 0x7B)  {
   //lowercase 'a' - 'z', so
   //convert to 'A' - 'Z'
     *psz_x = *psz_x - 0x20;
    }
   //advance to next char
    psz_x++;
   }
}

char sz_1[] = "Hello";
char sz_2[] = "UPPER/lower";

int main (void) {
  upcase(sz_1);
  upcase(sz_2);
}
```

In Assembly

```
sz_1:   .space  6 ;space for "hello",null
sz_2:   .space 12 ;space for "UPPER/lower",null
.text     ;Start of Code section
__reset:
    mov #__SP_init, W15    ;Init SP
    mov #__SPLIM_init,W0
    mov W0, SPLIM          ;Init SPLIM
    call init_variables    ;init strings
    rcall main             ;rcall main()
    reset                  ;start over
main:
    mov #sz_1,W0    ;W0 = &sz_1[0]
    rcall upcase
    mov #sz_2,W0    ;W0 = &sz_2[0]
    rcall upcase
done:
    goto    done      ;infinite wait loop

;*psz_x passed in W0
upcase:
   ...left as an exercise...
   return
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

What does ‘init_variables’ do?

• Initial values for variables live in program memory which is non-volatile.

• Can use a special mode called *program space visibility* (PSV) that allows upper half of memory to be mapped to program memory.

• Can then use instructions to copy data from program memory to data memory to initialize variables.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Init_variables

```
.text   ;program memory
;; constant data to be moved to data memory
sz_1_const: .asciz "Hello"
sz_2_const: .asciz "UPPER/lower"

init_variables:
;turn on program visibility space, use default PSVPAG value of 0
    bset CORCON,#2   ;enable PSV
;copy source address in program memory to W2
    mov  #psvoffset(sz_1_const),W2
    mov  #sz_1,W3       ;destination address in data memory
    rcall copy_cstring
;copy source address in program memory to W2
    mov  #psvoffset(sz_2_const),W2
    mov  #sz_2,W3       ;destination address in data memory
    rcall copy_cstring
    return
```

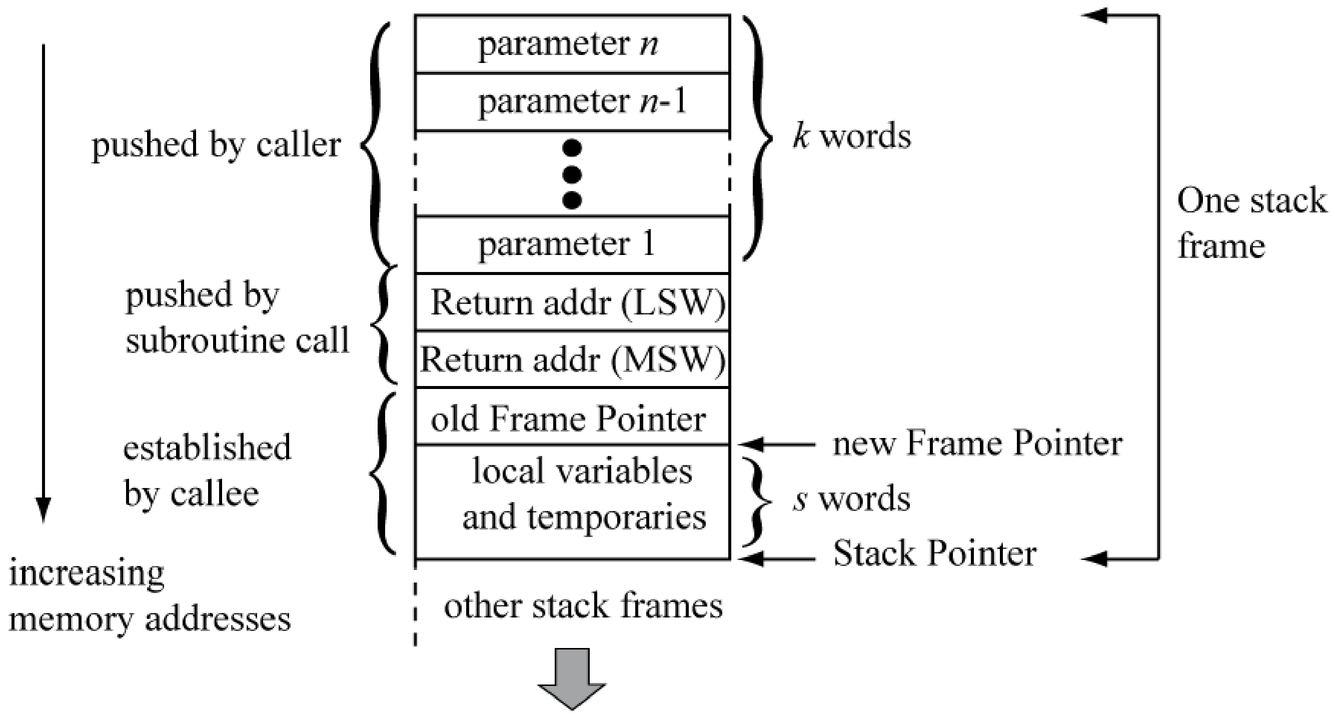
Copy strings from program memory to data memory

```
;;copy constant null-terminated string from program memory to data memory
;;W2 points to program memory, W3 to data memory
copy_cstring:
    mov.b [W2],W0
    cp.b W0,#0              ;test for null byte
    bra Z, copy_cstring_exit   ;exit if null byte
    mov.b [W2++],[W3++]       ;copy byte
    bra  copy_cstring         ;loop to top
copy_cstring_exit:
    mov.b [W2++],[W3++]       ;copy null byte
    return
```

Local variables versus global variables

- Global variables are assigned fixed memory locations in data memory by the compiler (i.e, 0x800, 0x802, etc).
- Local variables (variables declared in subroutines) are assigned either to working registers or to space allocated on the stack.
    - Allocation of stack space for local variables is called a ***stack frame***, and is an advanced topic that is not covered in this class (but is covered in the book and other slides). Stack frames can also be used to pass parameters to functions.
    - In our PIC24 implementations of C functions for homework or test problem examples, we will always use working registers for local variables declared in subroutines, and we will always use working registers to pass parameters to subroutines.

Stack Frames



Used when cannot fit locals, parameters in registers.
Use the stack for storage.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Constructing a Stack Frame

Caller

push parm_n
push parm_n–1
...... 1. Push parameters
push parm_1

`call` 2. Call subroutine

move 9. Save return value

*pops or SP = SP – (k*2)*
10. Clean stack of passed parameters

Subroutine

push FP 3. Save old frame pointer

SP → FP 4. Create new frame pointer, FP points at first local variable

*SP = SP + (s*2)* 5. Allocate local variable space

`lnk` instruction

do subroutine body, including establishing the return value.

FP → SP 6. Deallocate local variable space

pop FP 7. Restore old frame pointer

`ulnk` instruction

`return` 8. Return from subroutine

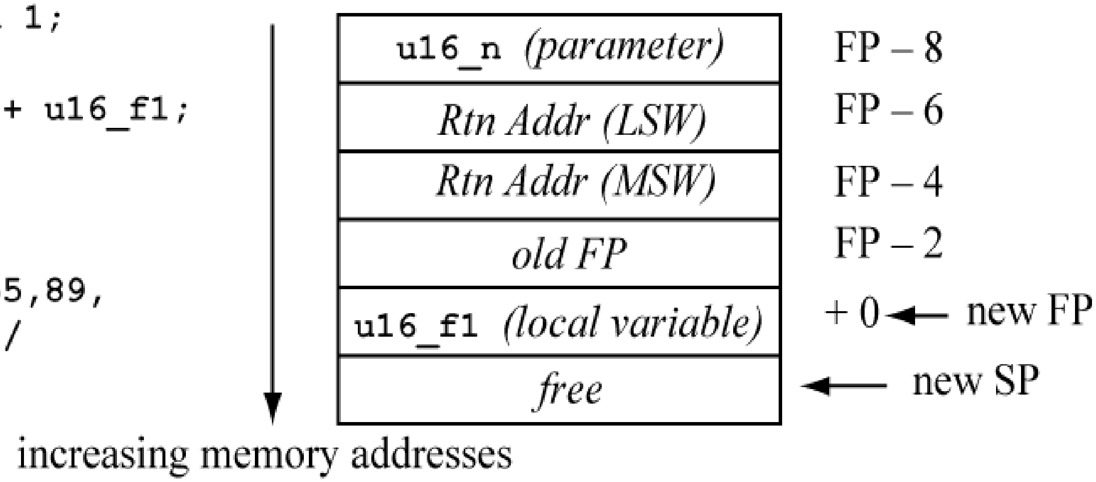
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Stack Frame for *fib()* Function

```
uint16 fib (uint16 u16_n) {
  uint16 u16_f1;

  if (u16_n == 0) return 0;
  if (u16_n == 1) return 1;
  u16_f1 = fib(u16_n-1);
  u16_f1 = fib(u16_n-2) + u16_f1;
  return (u16_f1);
}
/* Fib sequence:
0,1,1,2,3,5,8,13,21,34,55,89,
144,233,377,610,987,...*/
main (void) {
  uint16  u16_k, u16_j;
  u16_j = 13;
  u16_k = fib(u16_j);
  printf ("fib(%d) is: %d\n", u16_j, u16_k);
}
```

Detailed Stack Frame for `fib`



From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Calling fib()

In *C*

```
main (void) {
 uint16  u16_k, u16_j;
 u16_j = 13;

 u16_k = fib(u16_j);

 printf ("fib(%d) is: %d\n",
  u16_j, u16_k);
}
```

1. Push parameters
2. Call subroutine
9. Save return value
10. Clean stack of passed parameters

In Assembly

```
main:
 mov #13, W0
 mov WREG,u16_j      ;u16_j = 13
;subroutine call
 push W0             ;push u16_j on stack
 rcall fib
 mov WREG,u16_k
 sub W15,#2,W15      ;clean stack
                     ;of passed parameters

done:
    goto done        ;infinite wait loop
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

fib() Implementation

In C

```c
uint16 fib  (uint16 u16_n) {
  uint16 u16_f1;

  if (u16_n == 0) return 0;

  if (u16_n == 1) return 1;

  u16_f1 = fib(u16_n – 1);

  u16_f1 = fib(u16_n – 2)
        + u16_f1;

  return (u16_f1);
}
```

In Assembly

The `lnk` instruction does the following (3,4,5):
3. Save old FP (`push W14`)
4. Create new FP (`mov W15,W14`)
5. Allocate local space (`add W15,#2,W15`)

```
lnk #2

mov [W14-8],W0         ;access u16_n from FP(-8 offset)
cp W0,#0               ;u16_n == 0?
bra Z,fib_exit         ;exit with W0 = 0

cp W0,#1               ;u16_n == 1?
bra Z,fib_exit         ;exit with W0 = 1
;set up for next call
dec W0,W0              ;u16_n = u16_n - 1
push W0                ;push u16_n - 1 parameter
rcall fib              ;fib(u16_n - 1)
sub W15,#2,W15         ;clean u16_n - 1 parm. from stack
mov W0,[W14]           ;save returned u16_f1 in local
;set up for next call
mov [W14-8],W0         ;access u16_n from FP(-8 offset)
dec2 W0,W0             ;u16_n = u16_n - 2
push W0                ;save u16_n - 2 parameter
rcall fib              ;fib(u16_n - 2)
sub W15,#2,W15         ;clean u16_n - 2 parm. from stack
add W0,[W14],W0        ;W0 = fib(u16_n - 2) + u16_f1
fib_exit:

ulnk

return
```

The `ulnk` instruction does the following (6,7):
6. Deallocate local space (`mov W14,W15`)
7. Restore old frame pointer (`pop W14`)

8. Return from subroutine

fib() with stack frames required 20 instructions, without stack frames only 15 instructions. The generality of stack frames has overhead costs.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

What do you have to know?

- Indirect addressing modes for PIC24
- C code operation with pointers and arrays
- Implementation of C code with pointers/arrays in PIC24 assembly.
- How the stack on the PIC24 works
- How subroutine call/return works with the PIC24 stack
- How to pass parameters to a subroutine using registers for parameters and locals
- How to implement small C functions in PIC24 assembly.