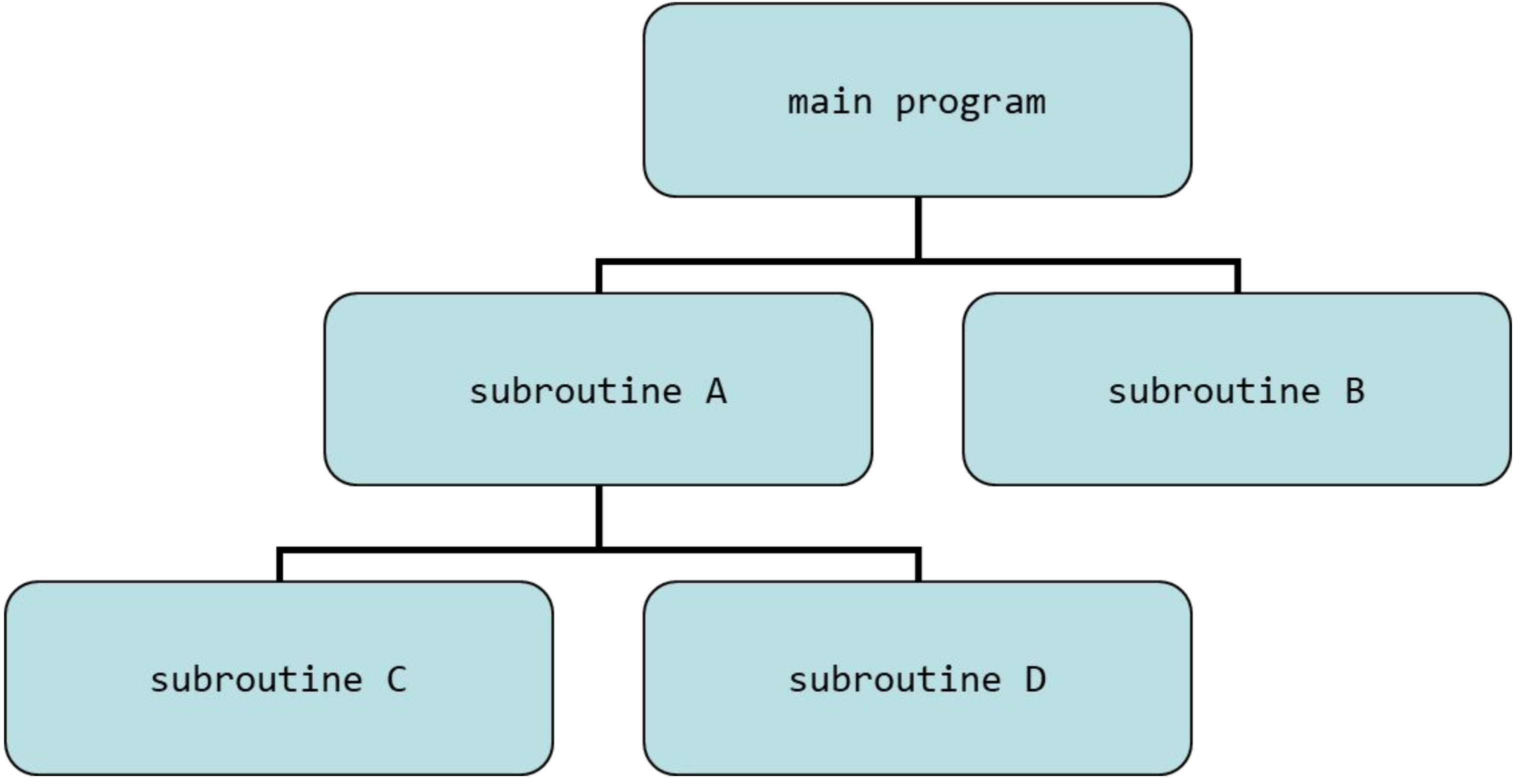
Subroutines

Textbook Chapter 7

Subroutines



Subroutines

- Kinds of subroutines:
  - value-returning → functions
  - non-value-returning → procedures (“void functions” in C/C++)
  - both will be referred to as subroutines
- In asm, a subroutine:
  - is a sequence of instructions
  - has a name (i.e. a labeled start address) → entry point
  - has a “return” instruction → exit point
- Multiple entry/exit points are possible, BUT NOT DESIRABLE

Subroutines

- What is the difference between a branch and a subroutine call?
  - Branch doesn't have to return to the calling location
  - A subroutine **must** return to the calling location
- How can this be done / What information is required?
  - The address to be returned to must be remembered.
- Where should this information be stored?
  - Multiple possibilities.

Subroutines - Where to Store Data?

- In a register
  - means that you lose the use of that register in the subroutine
  - How to handle a subroutine call in a subroutine: using the same register means you can't return to the initial call
  - Using additional registers reduces your number and easy to lose track

Subroutines - Where to Store Data?

- In a variable
  - means the caller has to create a variable the subroutine will use: how to know the name? what if name already in use?
  - If all subroutines use the same variable then how to handle a subroutine call in a subroutine

Subroutines - Store Data on the Stack

Steps for a subroutine call:

1. the address of the caller's next instruction (i.e. the return address) is pushed onto the stack
2. the PC is loaded with the subroutine start address

   … (the subroutine executes) …

3. the return address is popped off of the stack
4. the PC is loaded with the return address

Same problem as with branching: the programmer can't directly access the PC

Subroutines - Call a Subroutine

The 68000 provides two instructions for calling a subroutine:

```
jsr <ea>            ; jump to subroutine - analogous to jmp

bsr <ea>            ; branch to subroutine - analogous to bra
```

<ea> - the effective address will usually be a label

jsr <ea>

- jsr stands for **j**ump **s**ub**r**outine
- changes the SP and PC registers in the following way:

  SP $\leftarrow$ SP - 4

  (SP) $\leftarrow$ PC

  PC $\leftarrow$ address *operand is an absolute address*
- <ea> - the assembler translates the addressing mode into an address
- the changes are made in the exact order specified

bsr <ea>

- bsr stands for **b**ranch **s**ub**r**outine
- changes the SP and PC registers in the following way:

  SP $\leftarrow$ SP - 4

  (SP) $\leftarrow$ PC

  PC $\leftarrow$ PC + offset *operand is a relative displacement*
- <ea> - the assembler computes the distance from the instruction to the address - this is a displacement
- the changes are made in the exact order specified

Subroutine - Return from a Subroutine

The 68000 provides an instruction for returning from a subroutine:

```
rts return from subroutine
```

rts does:

$PC \leftarrow (SP)$

$SP \leftarrow SP + 4$

The changes are made in the exact order specified

Note: the CPU cannot determine if the top element of the stack is a valid address (all numbers are valid addresses!) so whatever value is there is used!

Subroutine - Example

```
NULL        equ   0
CR          equ   13
LF          equ   10
start: ...
            lea   str,a0
            jsr   write_string
...
str:        dc.b  "hello, world!",CR,LF,NULL
            even
; below could be in a separate source file:
write_string:
            move.b (a0)+,d0 ; takes input via a0
            beq   ws_exit
            bsr   write_char
            bra   write_string
ws_exit:    rts
write_char: ...             ; takes input via d0
            rts
```

Note: these are not acceptable subroutines for this course

Saving and Restoring Registers

- A subroutine will need to make use of a set of registers
    - its “working environment”.
- The caller also has a working environment.
- The caller’s working environment needs to be preserved across a subroutine call!

Saving and Restoring Registers

- Problem: in the example above, what if:
  - the main program needs a0 to remain pointing at the start of the string?
  - the main program is storing an important value in d0?
- The existing subroutine code corrupts both registers!

Solution for Storing/Restoring Registers

- Save the registers – on the stack
- Solution (C convention):
  - right after entry into a subroutine, save the value of each needed register
  - right before exit, restore the values
- Solution (Pascal Convention)
  - right before the call, the caller saves registers
  - right after the call, the caller restores saved registers

Solution for Storing/Restoring Registers

- Textbook uses Pascal method
  - Problem is what to save?
- If the calling code and the subroutine are written by different programmers – how is the caller to know what registers the subroutine uses?
  - Requires caller save all registers that are in use
  - More data than needed saved
- We will use the C convention in this class.

Example

```
my_subroutine:  move.w d0,-(sp) ; save original values (push)
                move.b d1,-(sp)
                move.l d2,-(sp)
                move.l a4,-(sp)
                ... ; code uses a4, d0 as word, d1 as byte, d2 as longword
                move.l (sp)+,a4 ; restore original values
                move.l (sp)+,d2 ; (pop in reverse order)
                move.b (sp)+,d1 ; ORDER is critical, pop must be done in
                move.w (sp)+,d0 ; in reverse to push
                rts         ; Personally I save and restore all used
                                ; registers as longwords
```

movem instruction

```
movem.<size> <register-list>,<ea>    ; save multiple registers
movem.<size> <ea>,<register-list>    ; restore multiple registers
```

```
my_subroutine:  movem.l d0-d2/d6/a4,-(sp)
                ...
                movem.l (sp)+,d0-d2/a4/d6
                rts
```

Order is irrelevant, but MUST have the same registers

Passing Input Parameters

There are three ways to do this:

1. Pass in registers
2. Pass in global variables
3. Pass on the stack
    - The correct way to pass parameters

Pass on the Stack

- The caller pushes each input parameter on to the stack
    - parameters are in a known order, before the call
- PROS:
    - quick & easy (once you get used to it)
    - unlimited number and size of parameters
    - good design: minimum coupling between caller & callee
    - reusable libraries possible
    - reentrant!
- CONS?

Methods of Parameter Passing

From 1701/1633 what methods are there for passing parameters?

- By value
  - Pass a copy of the value
  - In assembly you move a copy of the value onto the stack
  - This can be any data size
- By reference
  - Pass a reference (pointer) to the caller's variable
  - So in assembly you place the address of the variable on the stack
  - This can only be a longword
    - because it is an address

Calling Conventions

The C/C++ calling convention:

- parameters are pushed right to left
- caller cleans up the stack
- callee saves/restores the working environment

The alternative Pascal calling convention:

- parameters are pushed left to right
- callee cleans up the stack
- caller saves/restores the working environment
- The Pascal method is used in the textbook – far more prone to errors!

Calling Convention - Pass by Value

E.g. Consider the `write_char(chr)` subroutine. To call it in asm:

```
        move.b      chr,-(sp)         ; pass by value
        bsr     write_char
        addq.l      #2,sp             ; super pop, how much space added?
        …
chr:dc.b    ‘c’
```

Calling Convention - Pass by Reference

E.g. Consider the strcpy(char dst[],char src []) subroutine. To call it in asm:

```
        move.l      #src,-(sp) ; or pea src
        move.l      #dst,-(sp) ; or pea dst
        bsr strcpy
        addq.l      #8,sp ; super pop, how much space added
        …
src:dc.b        "copy me!",0
dst:ds.b        100
```

pea instruction

```
pea <ea>     ; push effective address onto the stack
```

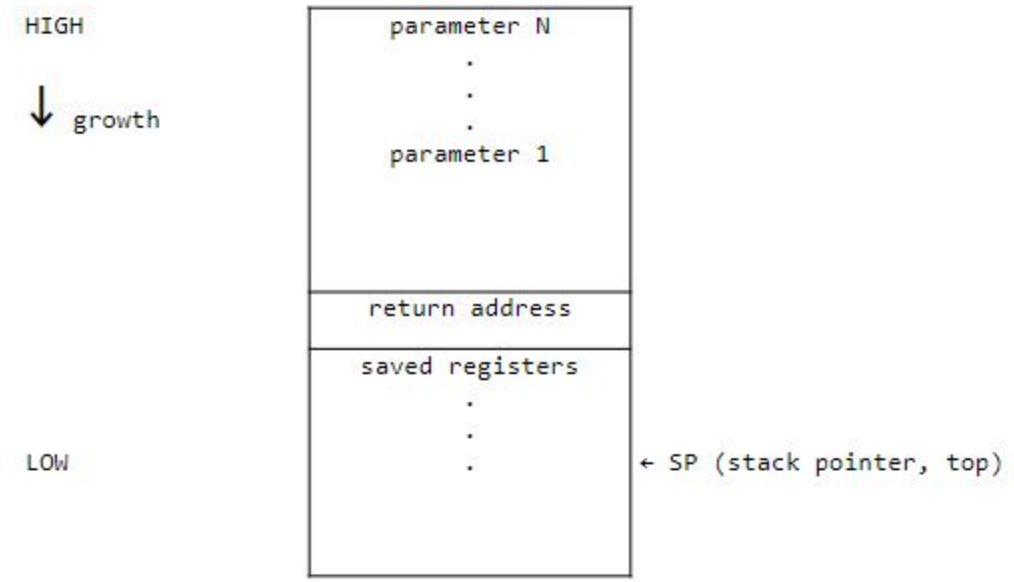
Is equivalent to the following two commands:

```
    lea     <ea>,Ai     ; Ai is any address register other than a7
    movea.l Ai,-(SP); note: A7 can be substituted for SP, why?
```

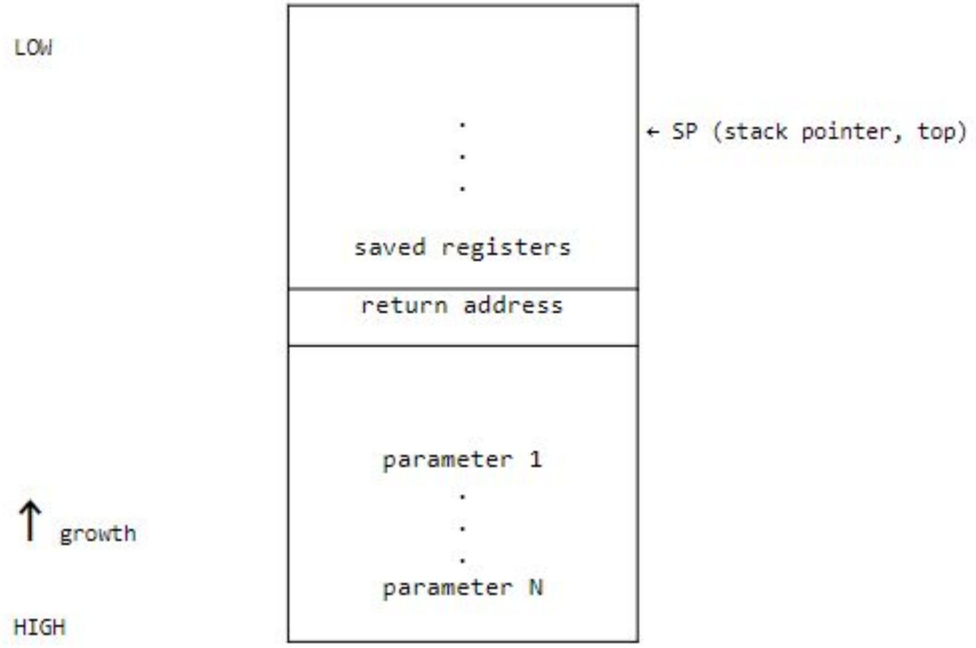
Example:

```
    pea wednesday        ; push the parameter address on the stack
    bsr subroutine       ; call the subroutine
    lea 4(sp),sp     ; why does this work instead of an addq.l
                         ; or adda.l
```

Stack Frame for Input Parameters



Stack Frame for Input Parameters



Accessing Parameters in a Stack Frame

To access an input parameter within a subroutine:

1. manually compute constant offset from a fixed position, which is the SP, based on:
    - # & size of saved registers
    - size of return address (4 bytes)
    - # & size of parameters
2. add offset to SP to get address using d16(An)
3. dereference

Example: void strcpy(char dst[], char src[])

	offset			
LOW				
	0	a1	L	← SP
	4	a0	L	
	8	return address	L	
↑ growth	12	dst	L	
HIGH	16	src	L	

Where does this stack frame layout come from based on what we know from the function prototype?

Parameter Offsets using the SP

```
; void strcpy(char dst[], char src[])
SC_DST       equ     12
SC_SRC       equ     16
strcpy: movem.l a0-a1,-(sp)
             move.l      SC_DST(sp),a0
             move.l      SC_SRC(sp),a1
             ; do copy here
             movem.l (sp)+,a0-a1
             rts
```

Issue with using SP

What if:

- We call a subroutine in a subroutine?
- Typically when this happens parameters to the initial subroutine as passed to the subsequent subroutine
- So after pushing one parameter then our offsets are incorrect: off by 4!

Solution: don’t base the offset on the stack pointer. Set up a frame pointer.

A different pointer to a fixed location in the current stack frame that doesn’t change!!

Could do this manually, but...

link instruction

`link An,#<displacement> set up An as a frame pointer`

This instruction:

- saves (pushes) the value of An on the stack
- loads An with SP
- adds the signed 16-bit displacement to SP

link instruction

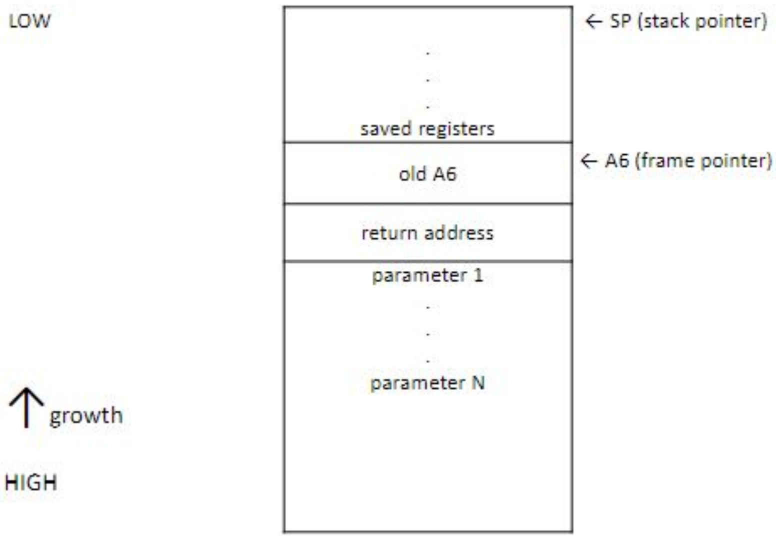
In other words:

```
SP      ← SP - 4
(SP)← An
An      ← SP
SP      ← SP + displacement
```

The register a_n is called the frame pointer. Typically, a_6 is used. Never use a_7! Why should you not use a_7 as your frame pointer?

For now use a displacement of ZERO

Stack Frame Becomes ...



Using link means that the first parameter will **always** be located at offset 8!

unlk instruction

```
unlk An ; clean up the frame pointer
```

This instruction does the opposite of link. In other words:

```
SP  ← An
An  ← (SP)
SP  ← SP + 4
```

Use the **same address register** that is used in the initial link instruction!

link/unlk example

```
my_subroutine:  link        a6,#0
                movem.l     d0-d2/a4,-(sp)
                ...
                movem.l (sp)+,d0-d2/a4
                unlk        a6
                rts
```

strcpy subroutine

```
SC_DST      equ     8
SC_SRC      equ     12
strcpy: link        a6,#0
            movem.l a0-a1,-(sp)
            move.l      SC_DST(a6),a0
            move.l      SC_SRC(a6),a1
sc_loop:    move.b      (a1)+,(a0)+
            bne     sc_loop
            movem.l (sp)+,a0-a1
            unlk a6
            rts
```

Stack Frame - void strcpy(char dst[], char src[])

	Offset			
LOW				
	-8	a1	L	← SP
	-4	a0	L	
	0	Old A6	L	← A6 framepointer
	4	return address	L	
↑ growth	8	dst	L	
HIGH	12	src	L	

Returning Output

- Like for input, the options are:
  - in registers (typically D0 or A0)
    - done in C/C++ for atomic values
    - not in 2655
  - in global variables
    - unacceptable
  - on the stack
    - preferred (mandatory in 2655)

Returning Output

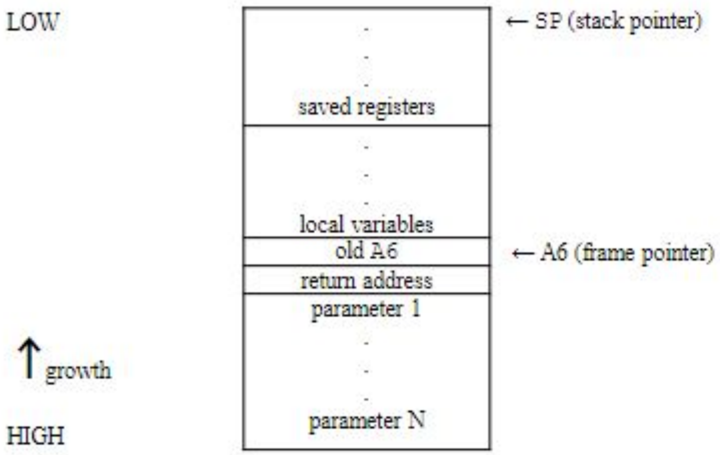
```
; call: int add_words(int x, int y); // assume int = 16 bits as:
; result = add_words(3, 39);
; subroutine will place return value in the given word on the stack
        subq.l  #2,sp               ; remember, stack grows down
        move.w  #39,-(sp)
        move.w  #3,-(sp)
        jsr     add_words
        addq.l  #4,sp               ; pop input parameters
        move.w  (sp)+,result(a6); pop output parameter
```

Stack Frame with Returning Output

	Offset			
LOW				
	-4	Saved environment		← SP
	0	Old A6	L	← A6 framepointer
	4	return address	L	
	8	3 (x)	W	
↑ growth	10	39 (y)	W	
HIGH	12	result	W	

Stack Frame for Local Variables

- Reserve space for local variables on the stack
- Use a **negative or zero** displacement in the link instruction
  - Allocation of space on the stack is always a subtraction



Subroutine with Local Variables Example

```
void foo(int x, int y) {
    int a;
    int b;
    a = x;
    b = x + y;
}
```

- Draw a picture of its stack frame, after its working environment has been saved
- Translate the definition to 68000 asm. Assume integers are 16 bits
    - Use named constants for all variable/parameter offsets

Assembly Code

```
FOO_X      equ        8
FOO_Y      equ        10
FOO_A      equ        -2
FOO_B      equ        -4
foo: link         a6,#0
     subq.l       #-2,sp          ; allocate stack space for a
     subq.l       #-2,sp          ; allocate stack space for b
     movem.l      d0,-(sp)        ; save environment being used
     move.w       FOO_Y(a6),d0
     move.w       FOO_X(a6),FOO_A(a6)
     move.w       FOO_X(a6),FOO_B(a6)
     add.w        d0,FOO_B(a6)
     movem.l      (sp)+,d0        ; restore environment that was used
     unlk         a6              ; why is there no code to deallocate local variables?
     rts
```

Stack Frame for Foo Subroutine

LOW	Offset			
		Saved environment		SP
	-4	b	W	
	-2	a	W	
	0	Old A6	L	A6 Framepointer
	4	return address	L	
growth	8	x	W	
HIGH	10	y	W	

link Instruction - Revisited

```
FOO_X     equ       8
FOO_Y     equ       12
FOO_A     equ       -2
FOO_B     equ       -4

foo:      link      a6,#-4
          movem.l   d0,-(sp)
          move.w    FOO_Y(a6),d0
          move.w    FOO_X(a6),FOO_A(a6)
          move.w    FOO_X(a6),FOO_B(a6)
          add.w     d0,FOO_B(a6)
          movem.l   (sp)+,d0
          unlk      a6
          rts
```