EE 357 Unit 7

Subroutines
Stacks

Subroutines

- Subroutines (or functions) are portions of code that we can call from anywhere in our code, execute that subroutine, and then return to where we left off

C code:

```
void main() {
   ...
   x = 8;
   res = avg(x,4);
   ...
}
int avg(int a, int b){
   return (a+b)/2;
}
```

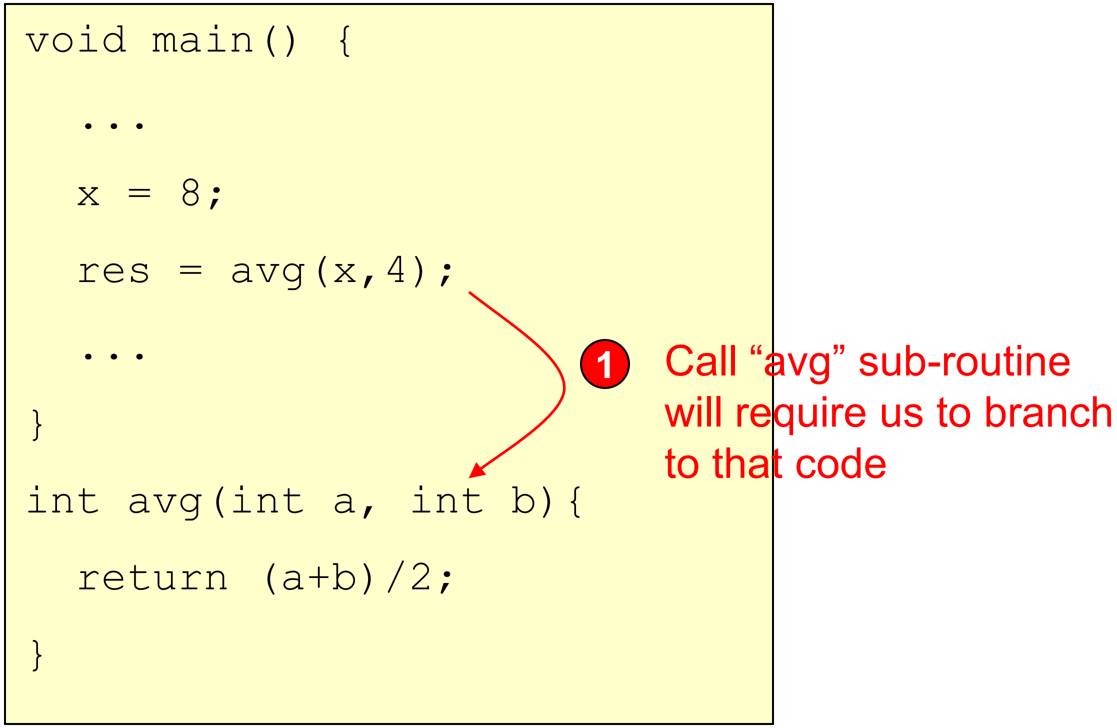
We call the subroutine to calculate the average

A subroutine to calculate the average of 2 numbers

Subroutines

- Subroutines are similar to branches where we jump to a new location in the code

C code:

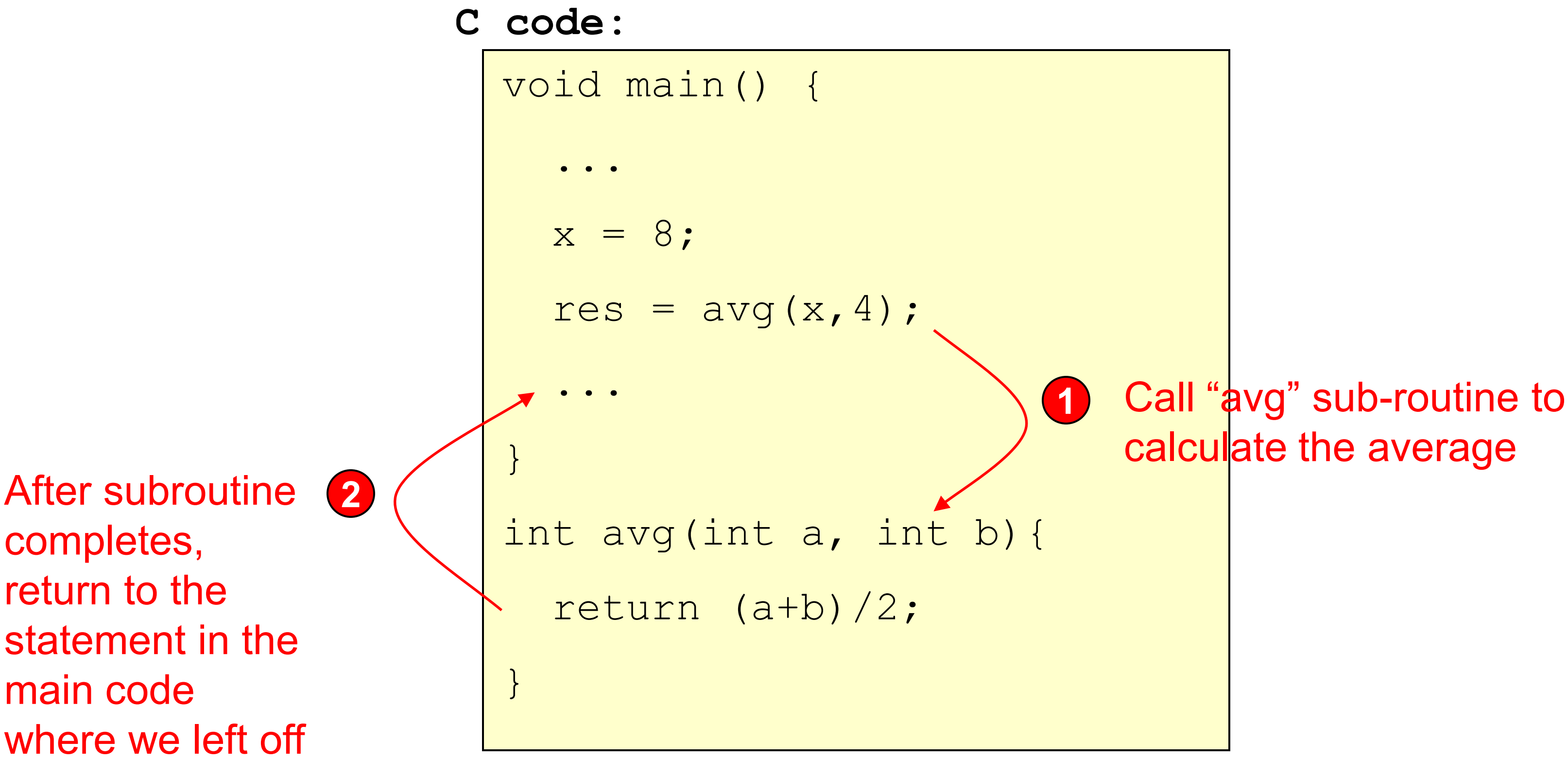


```
void main() {
   ...
   x = 8;
   res = avg(x,4);
   ...
}
int avg(int a, int b){
   return (a+b)/2;
}
```

Call “avg” sub-routine will require us to branch to that code

Normal Branches vs. Subroutines

- Difference between normal branches and subroutines is that subroutines automatically return to location after the subroutine call



C code:

```
void main() {
   ...
   x = 8;
   res = avg(x,4);
   ...
}
int avg(int a, int b){
   return (a+b)/2;
}
```

1 Call “avg” sub-routine to calculate the average

2 After subroutine completes, return to the statement in the main code where we left off

Implementing Subroutines

- To implement subroutines in assembly we need to be able to:
  - Branch to the subroutine code (BSR/JSR instruc.)
  - Return to the instruction after BSR when we finish the subroutine (RTS instruc.)

C code:

```
   ...
   res = avg(x,4);
   ...

int avg(int a, int b)
{ ... }
```

Assembly:

```
        ...
        BSR.W   AVG
        ...
        .org     0x0800
AVG:    ...
        RTS
```

Branching to a Subroutine

- Use BSR instruction (Branch SubRoutine)
- Format:
  - `BSR.{S,W,L}  Addr`
- Similar to branches we still add a displacement value to the PC
  [e.g. PC + disp→PC]
  - S,W,L refers to 8-, 16-, or 32-bit displacement values similar to normal branches
  - `Addr` is the address you want to branch to
    - *Usually specified as a label*
- Automatically stores the return address (RA) for use by the RTS instruction

BSR & RTS

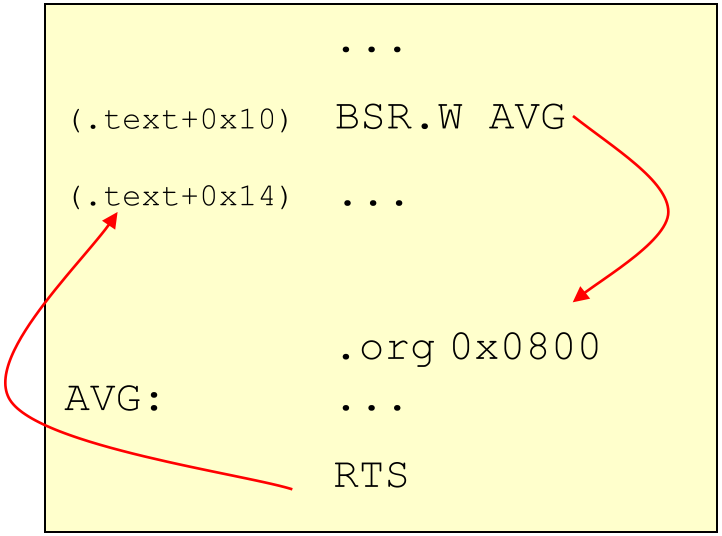
- Branch Formula:

  disp. = Addr of Label – (Addr. of Branch + 2)

  = 0x0800 – (0x0010+2) = 0x07EE

- Use RTS instruction to indicate that the subroutine is complete and we should return to where the routine was called

Assembly:



```
                ...
(.text+0x10)    BSR.W  AVG
(.text+0x14)    ...


                .org 0x0800
AVG:            ...

                RTS
```

1 BSR will add displacement 0x07EE to PC to get the new PC = .text+0x0800

2 BSR will also store the return address: .text+0x14 for use by the RTS instruction

3 RTS loads the PC with the return address stored by the BSR

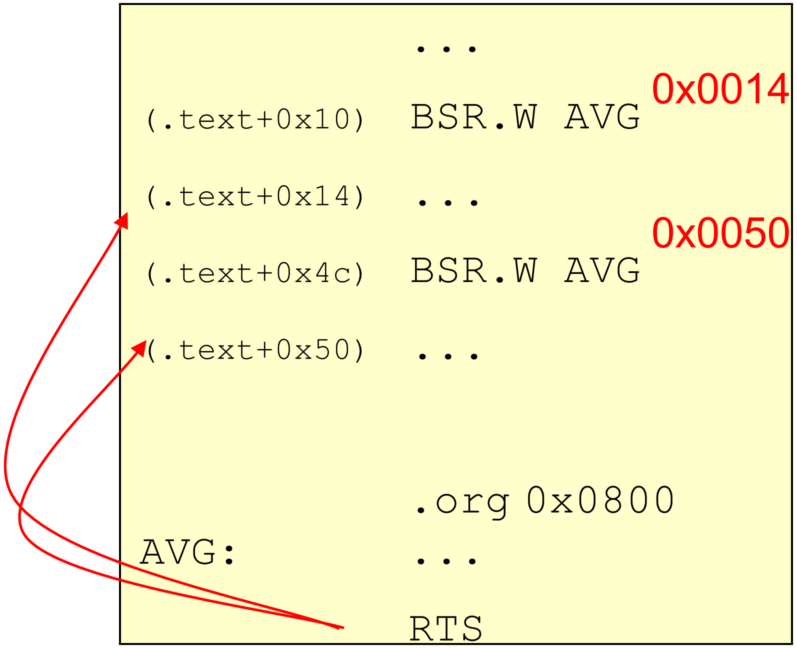
Jumping to a Subroutine

- Format:
  - `JSR   Addr`
- Rather than storing a displacement to add to the PC like BSR does, JSR simply stores the start address of the subroutine
  [e.g. PC = Addr.]
  - `Addr` is the address you want to branch to
    - *Usually specified as a label*
- Automatically stores the return address (RA) for use by the RTS instruction just like BSR

Return Addresses: Fact 1

- AVG may be called many times from many places in the code…which means a different return address each time

Assembly:



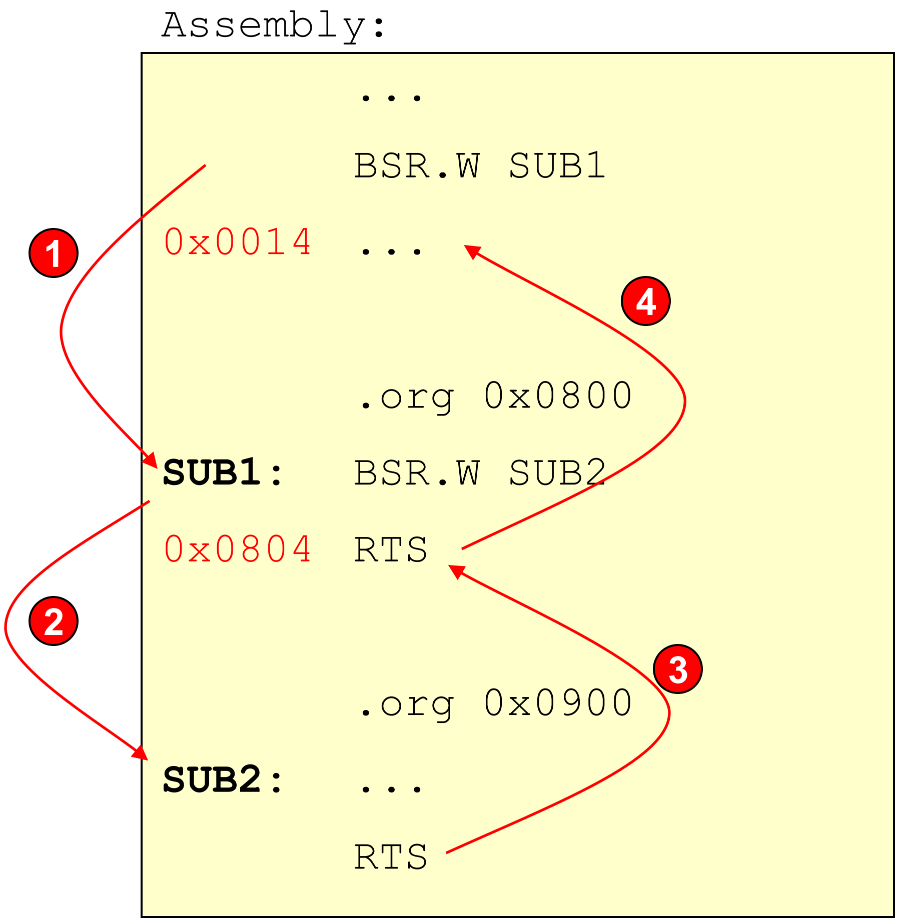
```
                ...
(.text+0x10)    BSR.W  AVG
(.text+0x14)    ...
(.text+0x4c)    BSR.W  AVG
(.text+0x50)    ...


                .org 0x0800
AVG:            ...
                RTS
```

0x0014 is the return address for this BSR

0x0050 is the return address for this BSR

Return Addresses: Fact 2

- Subroutines may call other subroutines (i.e. arbitrary number of “nested” subroutines calls)
- Example: ‘main’ calls ‘sub1’ which calls ‘sub2’ and so on…
  - Need to store all these return addresses until they are used by RTS instructions



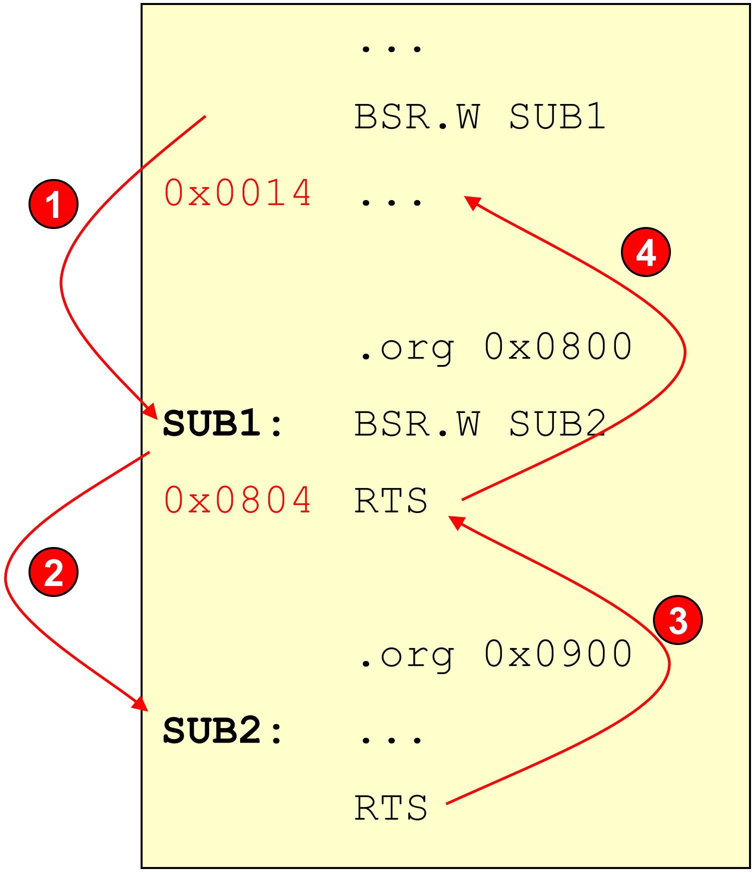
Assembly:

```
          ...
          BSR.W SUB1
0x0014    ...

          .org 0x0800
SUB1:     BSR.W SUB2
0x0804    RTS

          .org 0x0900
SUB2:     ...
          RTS
```

Dealing with Return Addresses

- Q: Can we store return address in a particular processor register (i.e. select A6 as the “return address register”)?
- A: No because we have multiple return addresses alive at the same time
- Q: Can we use more registers to store RA’s?
- A: Not if we want to support arbitrary depth of subroutine calls

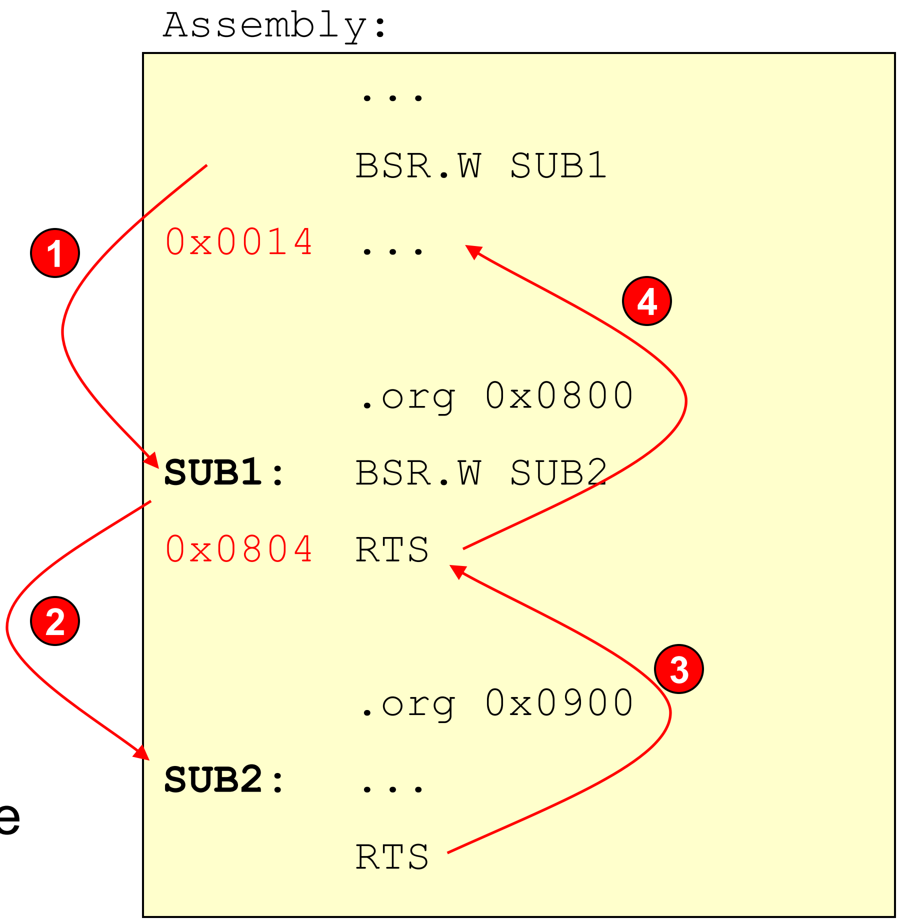
Assembly:



```
        ...
        BSR.W  SUB1
0x0014  ...

        .org  0x0800
SUB1:   BSR.W  SUB2
0x0804  RTS

        .org  0x0900
SUB2:   ...
        RTS
```

Storing Return Addresses

- To store arbitrary number of return addresses, need to store them in memory (i.e. usually enough main memory to allow for arbitrary depth of subroutine calls)
- Return addresses will be accessed in reverse order as they are stored
  - 0x0804 is the second RA to be stored but should be the first one used to return
- Implies we should use a stack data structure to store RA's

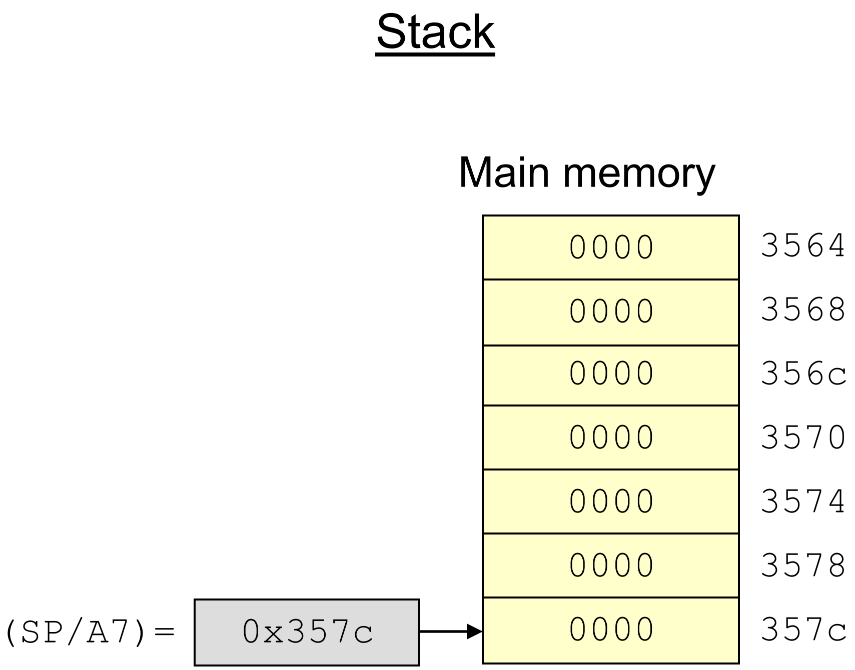


Assembly:

```
        ...
        BSR.W SUB1
0x0014  ...

        .org 0x0800
SUB1:   BSR.W SUB2
0x0804  RTS

        .org 0x0900
SUB2:   ...
        RTS
```

Stacks

- Use a stack to store the return addresses
- Stack is a data structure where data is accessed in reverse order as it is stored
- System stack will use a specific area in memory and growing towards smaller addresses
- Stack is accessed using A7 as a pointer
  - A7 is renamed SP (Stack pointer)



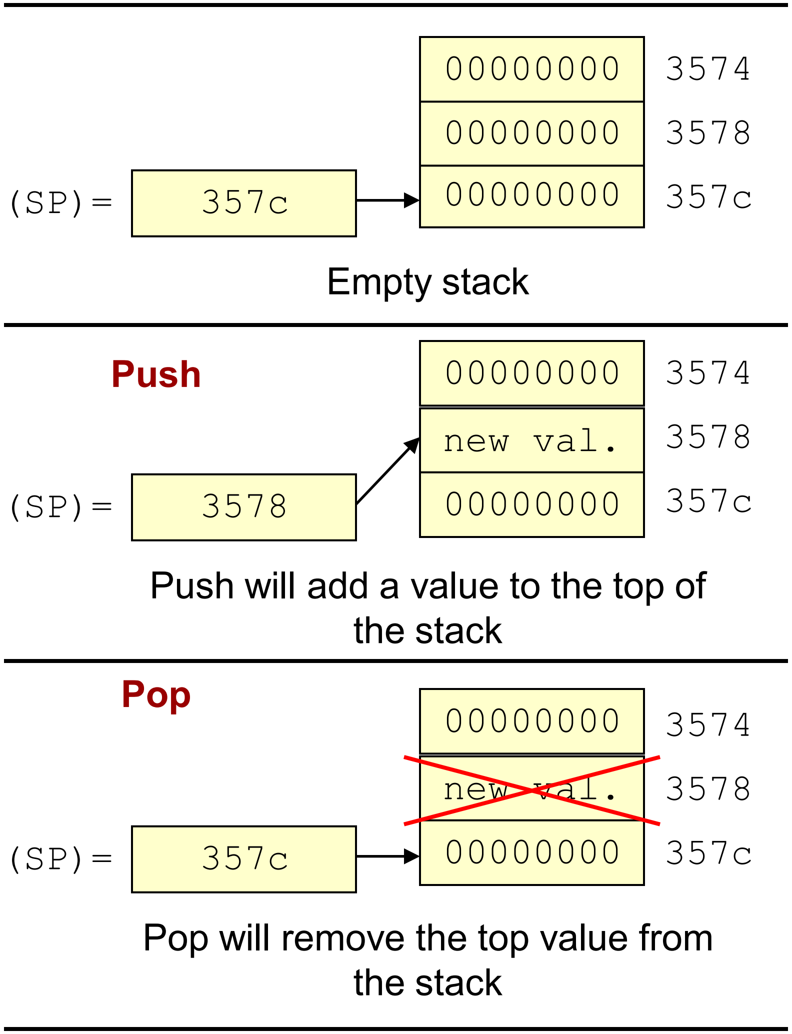
0000	3564
0000	3568
0000	356c
0000	3570
0000	3574
0000	3578
0000	357c

Stack Pointer Convention
Always points to top occupied element of the stack

0x0357c is the base of the stack, but it will always be empty due to our convention

Stacks

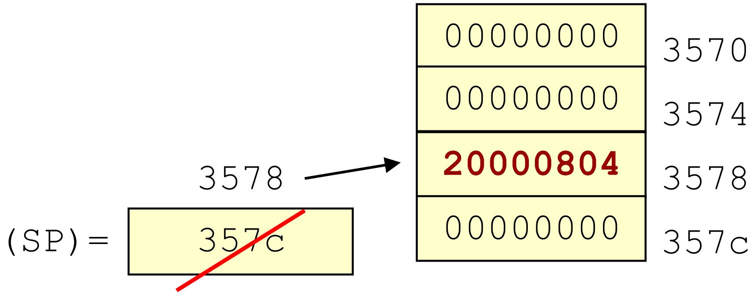
- 2 Operations on stack
  - Push: Put new data on top of stack
    - Decrement SP
    - Write value to where SP points
  - Pop: Retrieves and “removes” data from top of stack
    - Read value from where SP points
    - Increment SP to effectively “delete” top value



Push Operation

- Push: Put new data on top of stack
  - Decrement SP
    - SP = SP-4
  - Write value to where SP points
    - M[SP] = value
  - Can be accomplished w/ predecrement mode
    - MOVE.L D0,-(SP)

Push value 0x20000804

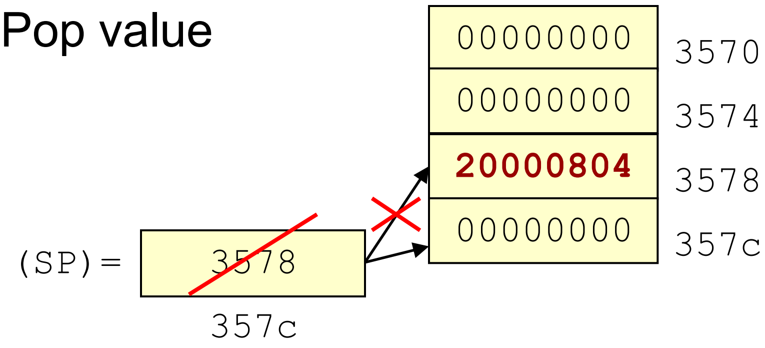


Decrement SP by 4 (since pushing a longword), then write value to where the SP is now pointing

Pop Operation

- Pop: Retrieves and "removes" data from top of stack
  - Read value from where SP points
    - dst = M[SP]
  - Increment SP to effectively "delete" top value
    - SP = SP + 4
  - Can be accomplished w/ predecrement mode
    - MOVE.L (SP)+,D0

Pop value



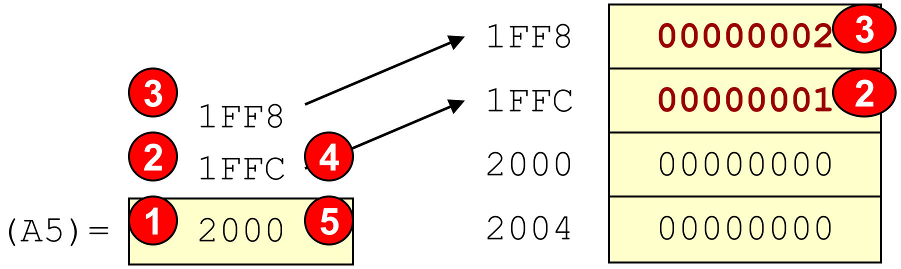
Read value that SP points at then increment SP (this effectively deletes the value because the next push will overwrite it)

***Warning**: Because the stack grows towards lower addresses, when you push something on the stack you subtract 4 from the SP and when you pop, you add 4 to the SP.*

User-Defined Stack Example

- Users can create their own stack data structures using M68000 Instructions
  - Predecrement mode is perfect for push operations while postincrement mode is perfect for pop operations
    - Due to the assumption that top of stack is 1st occupied location and that the stack grows upwards (towards lower addresses)

```
* Setup stack pointer
MOVEA.L #$2000,A5        1
* Use pre-decrement to push
MOVE.L  #1,-(A5)         2
MOVE.L  #2,-(A5)         3
* Use post-increment to pop
MOVE.L  (A5)+,D0         4
MOVE.L  (A5)+,D1         5
```



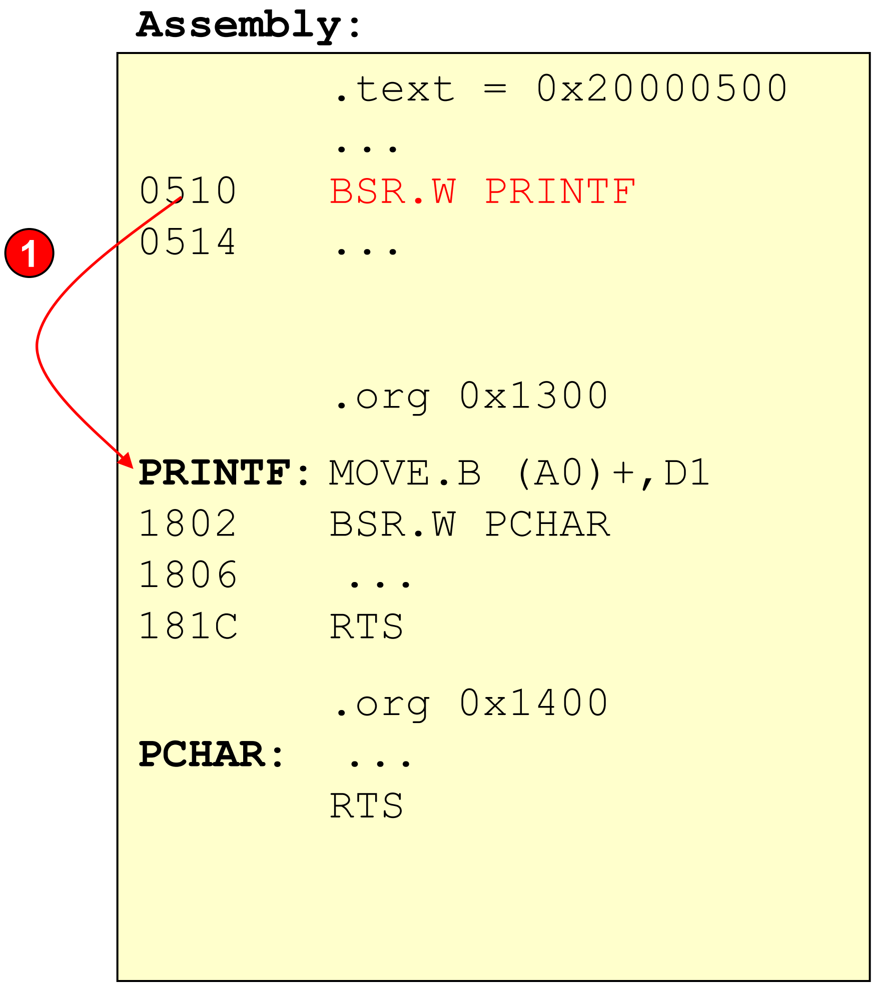
M68000 System Stack

- RA’s are used in reverse (LIFO) order
- Processor “automatically” maintains system stack
  - A7 is used as system stack pointer
    - Points to top occupied element of stack
    - Aliased with name SP [i.e. use (SP)+, -(SP), etc.]
  - Coldfire 5211 initializes SP/A7 to 0x20004000
  - In this class we will start our stack at 0x357C (just to use save writing).
- BSR/JSR will automatically PUSH the RA
- RTS will automatically POP the RA

Subroutines and Stacks

- BSR instruction automatically pushes the RA
  - – SP = SP - 4
  - – M[SP] = Return Address
  - – PC = PC + displacement
- RTS instruction automatically pops the RA
  - – PC = M[SP] // i.e. return address put back in PC
  - – SP = SP + 4

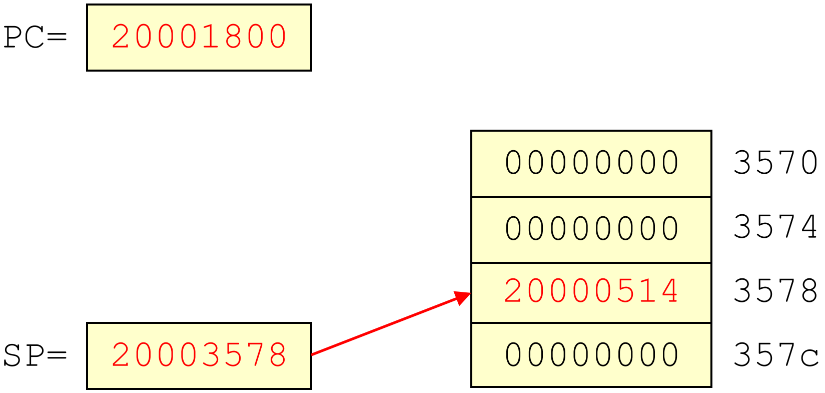
An Example



Assembly:

```
         .text = 0x20000500
         ...
0510     BSR.W PRINTF
0514     ...


         .org 0x1300
PRINTF:  MOVE.B (A0)+,D1
1802     BSR.W PCHAR
1806      ...
181C     RTS

         .org 0x1400
PCHAR:    ...
         RTS
```

- When we hit the first BSR, it will push the RA on the stack and update the PC to 20001800



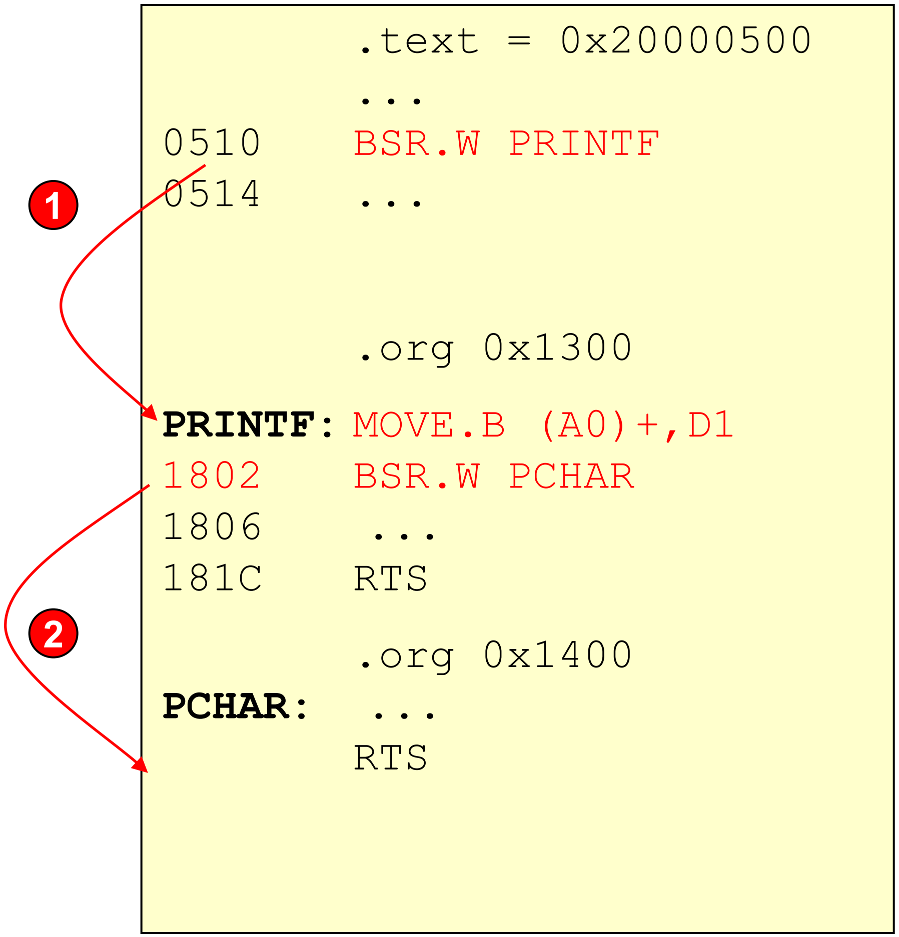
PC= 20001800

SP= 20003578

00000000	3570
00000000	3574
20000514	3578
00000000	357c

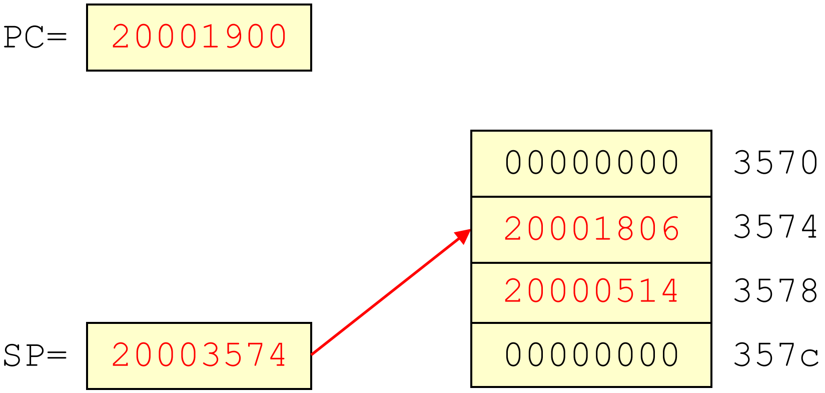
An Example

Assembly:



```
          .text = 0x20000500
          ...
0510      BSR.W PRINTF
0514      ...


          .org 0x1300
PRINTF:   MOVE.B (A0)+,D1
1802      BSR.W PCHAR
1806       ...
181C      RTS

          .org 0x1400
PCHAR:     ...
          RTS
```

- When we hit the second BSR, it will push the RA on the stack and update the PC to 0x20001900

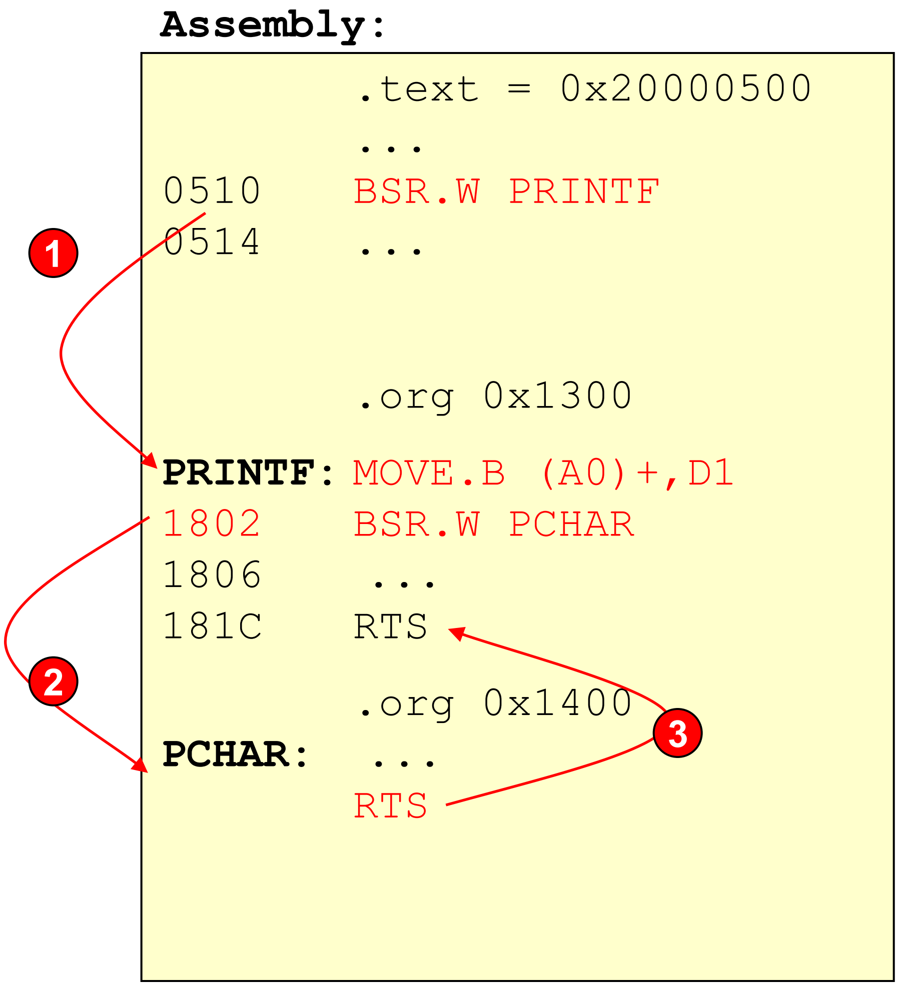


PC= 20001900

SP= 20003574

00000000	3570
20001806	3574
20000514	3578
00000000	357c

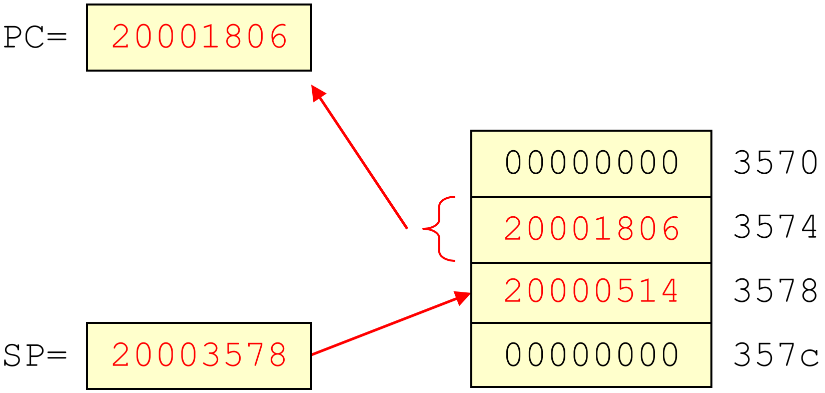
An Example



Assembly:

```
            .text = 0x20000500
            ...
0510        BSR.W PRINTF
0514        ...


            .org 0x1300
PRINTF:     MOVE.B (A0)+,D1
1802        BSR.W PCHAR
1806         ...
181C        RTS

            .org 0x1400
PCHAR:       ...
            RTS
```

- The first RTS will pop off the RA from the top of the stack and return the PC to 0x20001806

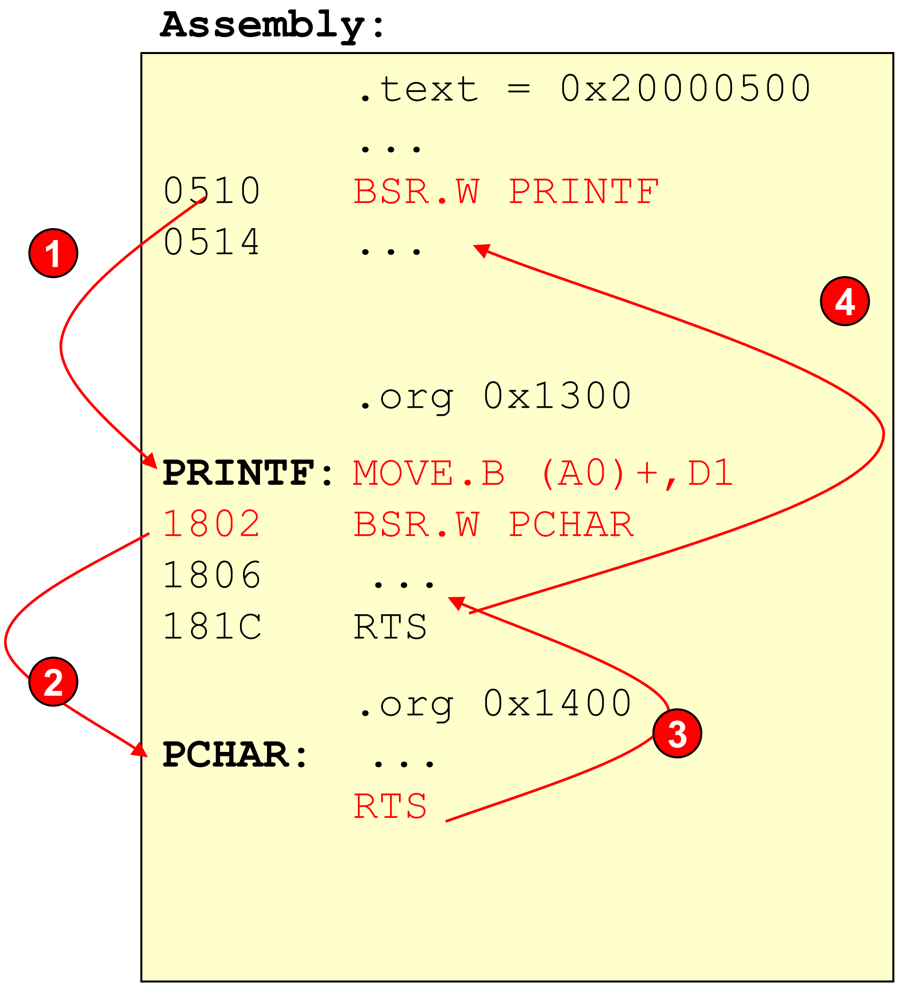


PC= 20001806

SP= 20003578

00000000	3570
20001806	3574
20000514	3578
00000000	357c

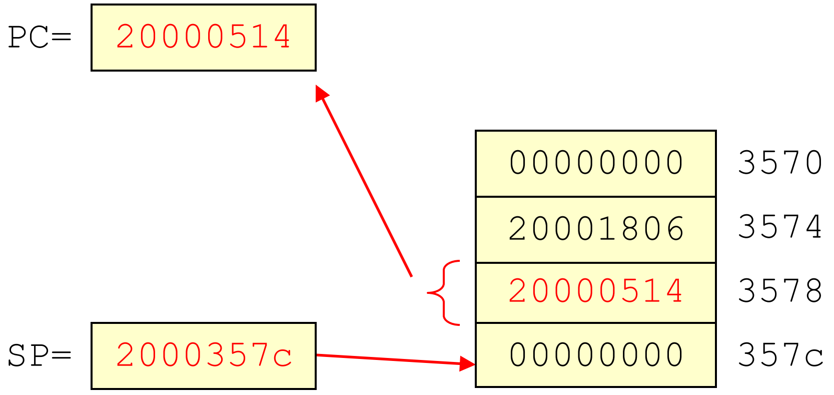
An Example



Assembly:

```
        .text = 0x20000500
        ...
0510    BSR.W PRINTF
0514    ...


        .org 0x1300
PRINTF: MOVE.B (A0)+,D1
1802    BSR.W PCHAR
1806     ...
181C    RTS

        .org 0x1400
PCHAR:   ...
        RTS
```

- The second RTS will pop off the RA from the top of the stack and return the PC to 0x20000514



PC= 20000514

SP= 2000357c

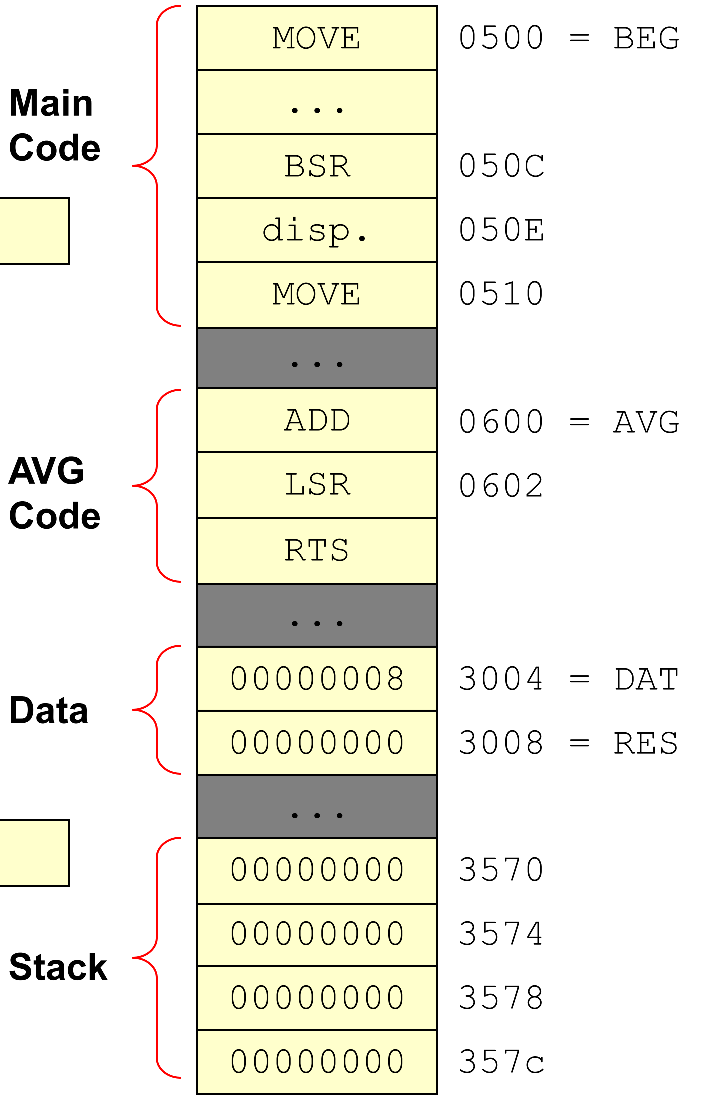
00000000	3570
20001806	3574
20000514	3578
00000000	357c

Subroutine Example

```
        .data
DAT:    .long   8
RES:    .space  4

        .text
BEG:    MOVE.L  DAT,D0
        MOVE.L  #4,D1
        BSR.W   AVG
        MOVE.L  D1,RES
        ...

        .org    0x100
AVG:    ADD.L   D0,D1
        LSR.L   #1,D1
        RTS
```

PC= 0500

D0= 00000000   D1= 00000000   SP= 357c



Main Code	MOVE	0500 = BEG
	...	
	BSR	050C
	disp.	050E
	MOVE	0510
	...	
AVG Code	ADD	0600 = AVG
	LSR	0602
	RTS	
	...	
Data	00000008	3004 = DAT
	00000000	3008 = RES
	...	
Stack	00000000	3570
	00000000	3574
	00000000	3578
	00000000	357c

- .text section starts at 0x500
  .data section starts at 0x3004
  Stack starts at 0x357c and grows upward

Subroutine Example

```
        .data
DAT:    .long   8
RES:    .space  4

        .text
BEG:    MOVE.L  DAT,D0
        MOVE.L  #4,D1
        BSR.W   AVG
        MOVE.L  D1,RES
        ...

        .org    0x100
AVG:    ADD.L   D0,D1
        LSR.L   #1,D1
        RTS
```

PC= 050C

D0= 00000008

D1= 00000004

SP= 357c

MOVE	0500 = BEG
...	
BSR	050C
disp.	050E
MOVE	0510
...	
ADD	0600 = AVG
LSR	0602
RTS	
...	
00000008	3004 = DAT
00000000	3008 = RES
...	
00000000	3570
00000000	3574
00000000	3578
00000000	357c

- First two move instructions initialize D0 and D1 with 8 and 4 respectively

Subroutine Example

```
        .data
DAT:    .long   8
RES:    .space  4

        .text
BEG:    MOVE.L  DAT,D0
        MOVE.L  #4,D1
        BSR.W   AVG
        MOVE.L  D1,RES
        ...

        .org    0x100
AVG:    ADD.L   D0,D1
        LSR.L   #1,D1
        RTS
```

PC= 0600

D0= 00000008

D1= 00000004

SP= 3578

MOVE	0500 = BEG
...	
BSR	050C
disp.	050E
MOVE	0510
...	
ADD	0600 = AVG
LSR	0602
RTS	
...	
00000008	3004 = DAT
00000000	3008 = RES
...	
00000000	3570
00000000	3574
20000510	3578
00000000	357c

- BSR pushed return address of 0x0510 onto stack and sets PC to 0x0600 (AVG subroutine)

Subroutine Example

```
        .data
DAT:    .long   8
RES:    .space  4

        .text
BEG:    MOVE.L  DAT,D0
        MOVE.L  #4,D1
        BSR.W   AVG
        MOVE.L  D1,RES
        ...

        .org    0x100
AVG:    ADD.L   D0,D1
        LSR.L   #1,D1
        RTS
```

PC= 0604

D0= 00000008

D1= 00000006

SP= 3578

MOVE	0500 = BEG
...	
BSR	050C
disp.	050E
MOVE	0510
...	
ADD	0600 = AVG
LSR	0602
RTS	
...	
00000008	3004 = DAT
00000000	3008 = RES
...	
00000000	3570
00000000	3574
20000510	3578
00000000	357c

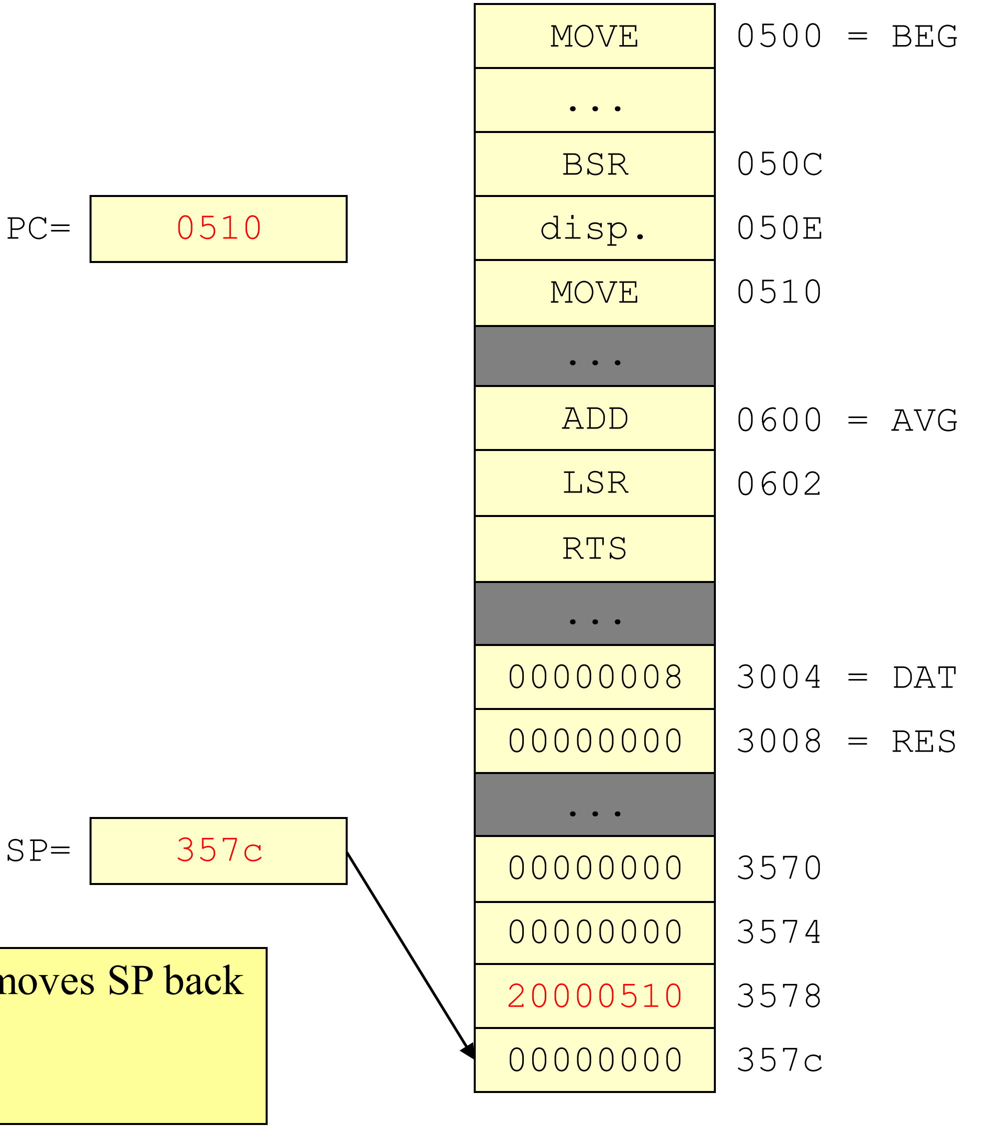
- ADD and LSR instructions find average of 8 and 4 which is 6.

Subroutine Example

```
        .data
DAT:    .long   8
RES:    .space  4

        .text
BEG:    MOVE.L  DAT,D0
        MOVE.L  #4,D1
        BSR.W   AVG
        MOVE.L  D1,RES
        ...

        .org    0x100
AVG:    ADD.L   D0,D1
        LSR.L   #1,D1
        RTS
```

D0= 00000008

D1= 00000006



PC= 0510

SP= 357c

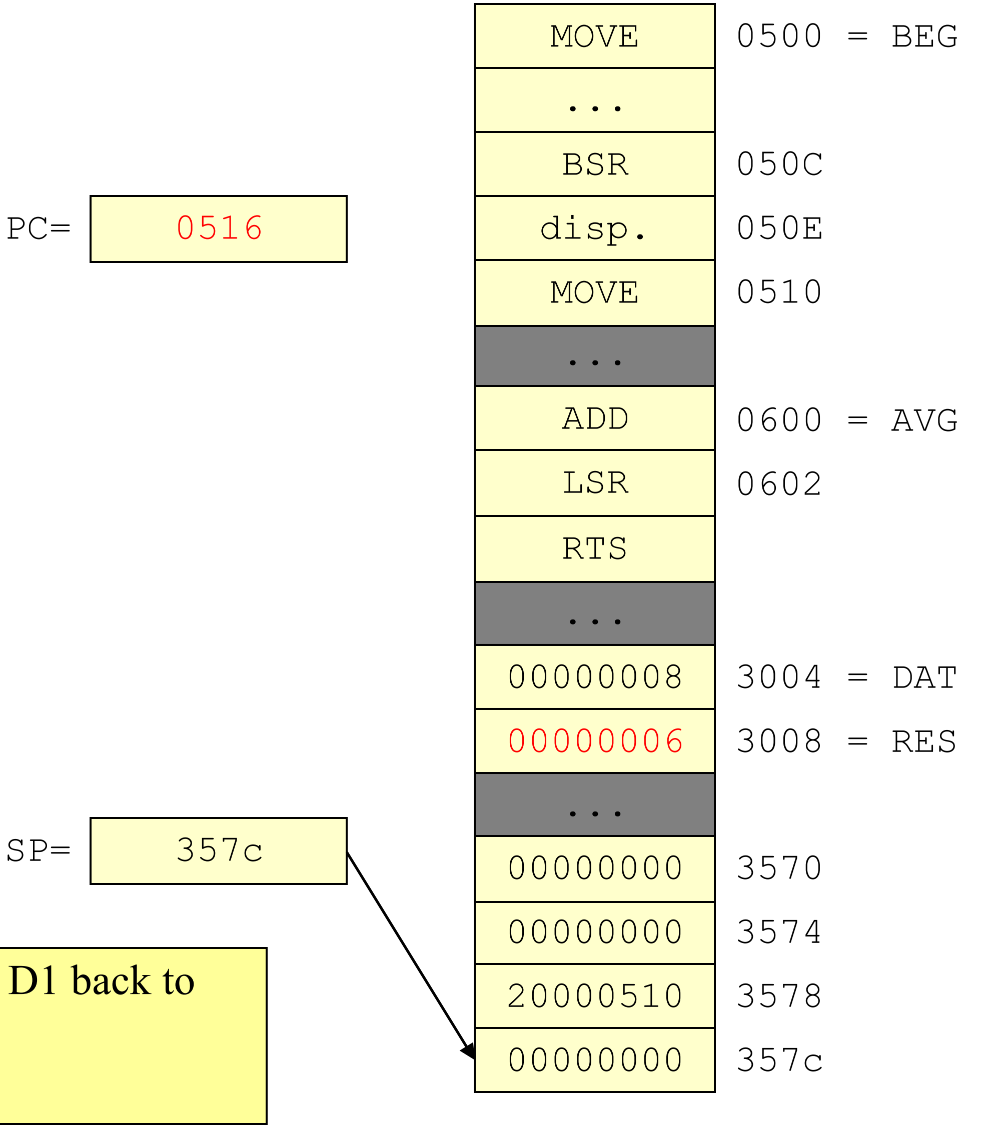
MOVE	0500 = BEG
...	
BSR	050C
disp.	050E
MOVE	0510
...	
ADD	0600 = AVG
LSR	0602
RTS	
...	
00000008	3004 = DAT
00000000	3008 = RES
...	
00000000	3570
00000000	3574
20000510	3578
00000000	357c

- RTS pops RA (0x0510) back into PC and moves SP back down to 0x357c

Subroutine Example

```
        .data
DAT:    .long   8
RES:    .space  4

        .text
BEG:    MOVE.L  DAT,D0
        MOVE.L  #4,D1
        BSR.W   AVG
        MOVE.L  D1,RES
        ...

        .org    0x100
AVG:    ADD.L   D0,D1
        LSR.L   #1,D1
        RTS
```

D0= 00000008   D1= 00000006



PC= 0516

SP= 357c

MOVE	0500 = BEG
...	
BSR	050C
disp.	050E
MOVE	0510
...	
ADD	0600 = AVG
LSR	0602
RTS	
...	
00000008	3004 = DAT
00000006	3008 = RES
...	
00000000	3570
00000000	3574
20000510	3578
00000000	357c

- MOVE instruction writes average result in D1 back to memory
  Notice stack is back to original position

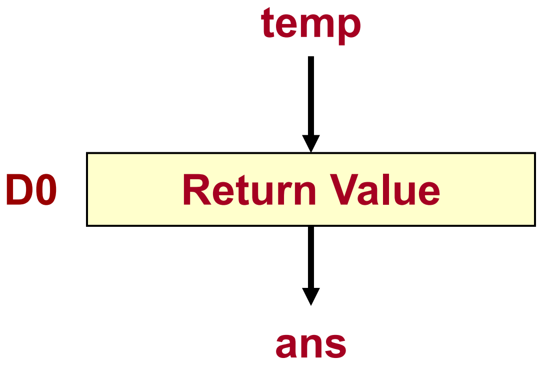
Subroutine Arguments & Return Values

- Hand-coded assembly
  - Parameters and return values can be passed/returned in specific registers
    - If too many arguments, use the stack (i.e. caller can push arguments on and have callee retrieve them
  - Passing values in registers does not work for recursive of re-entrant routines where multiple instances of the routine can be running at the same time => Need separate storage for each instance
- Compilers
  - Almost always use the stack
  - Create a structure on the stack known as a “frame”

Return Values

- Subroutines often need to return a value
- HLL's like C only allow 1 return value
  - Usually returned in a specific register
  - For Coldfire, D0 is usually the return value

```
int ans;
void main() {
  ans = avg(1,5);
}

int avg(int a, int b) {
  int temp = 1;
  return a + b >> temp;
}
```



temp

D0

Return Value

ans

Saving Registers

- One routine may generate values in registers, call a subroutine, and then expect to use those values
- Meanwhile the subroutine may overwrite the register
- Solution: Save registers on the stack before overwriting them in the subroutine, then restore them before returning

```
        .text
main:   move.l   var,d5         1
        movea.l #dat,a0
        ...
        bsr.w    sub1
        ...
        add.l    (a0)+,d5       3

        .org 0x0300
sub1:   move.l #1,d5            2
        ...
        rts
```

```
        .text
main:   move.l   var,d5
        movea.l #dat,a0
        ...
        bsr.w    sub1
        ...
        add.l    (a0)+,d5

        .org 0x0300
sub1:   move.l d5,-(sp)
        move.l #1,d5
        ...
        move.l (sp)+,d5
        rts
```

Recursive Factorial Routine

C Code:

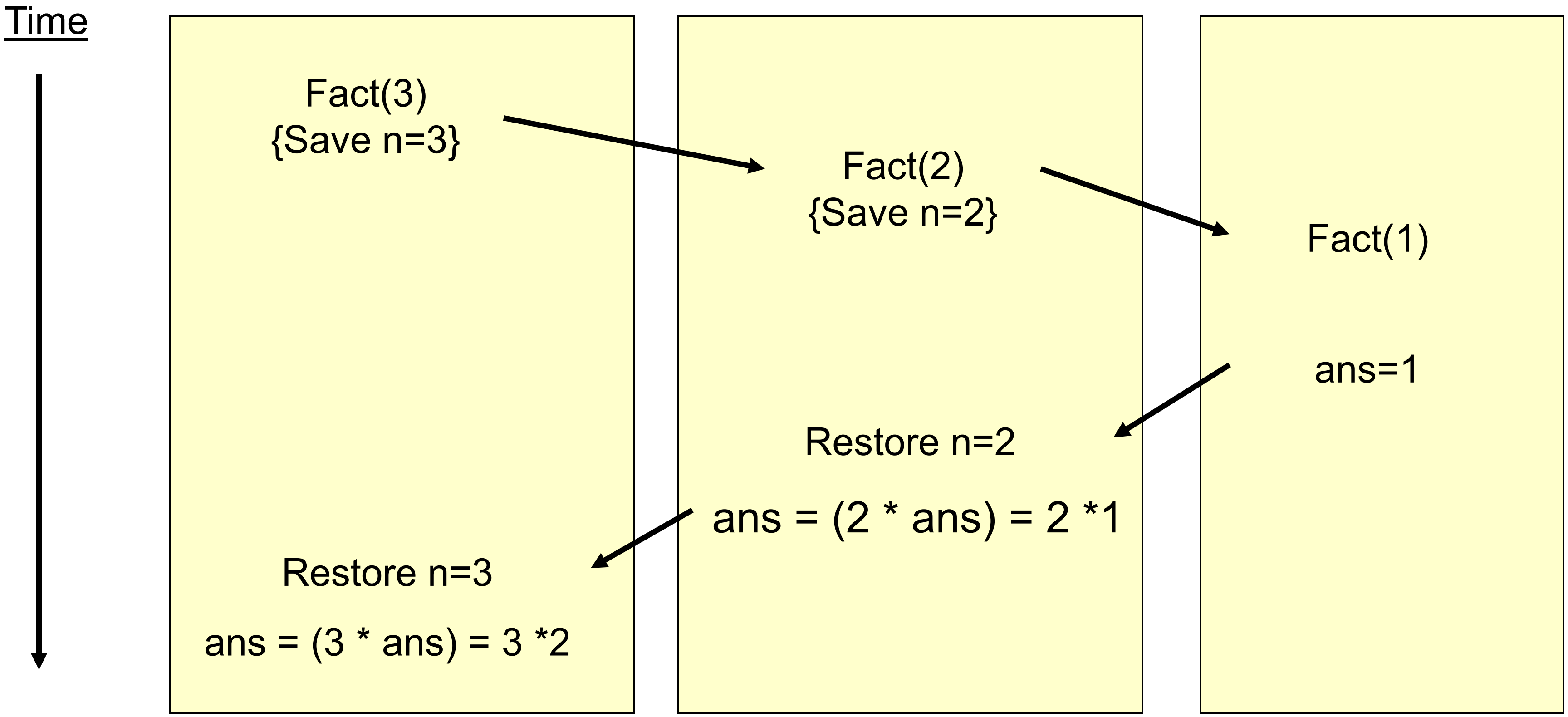
```
int ans;

void fact(int n)
 {
   if(n == 1)
       ans = 1;
   else {
      // calculate (n-1)!
      fact(n-1);

      // now ans = (n-1)!
      // so calculate n!
      ans = n * ans;

   }
 }
```

Assembly:

```
            .data
ANS:        .space    4

            .text
START:      MOVE.L    #3,D0
            BSR.W     FACT
            MOVE.L    D1,ANS
            STOP      #$2700

            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
            MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```

- Implementation Detail: Make sure each call of Fact is working with its own value of n

Recursive Call Timeline



Time

Fact(3)
{Save n=3}

Fact(2)
{Save n=2}

Fact(1)

ans=1

Restore n=2

ans = (2 * ans) = 2 *1

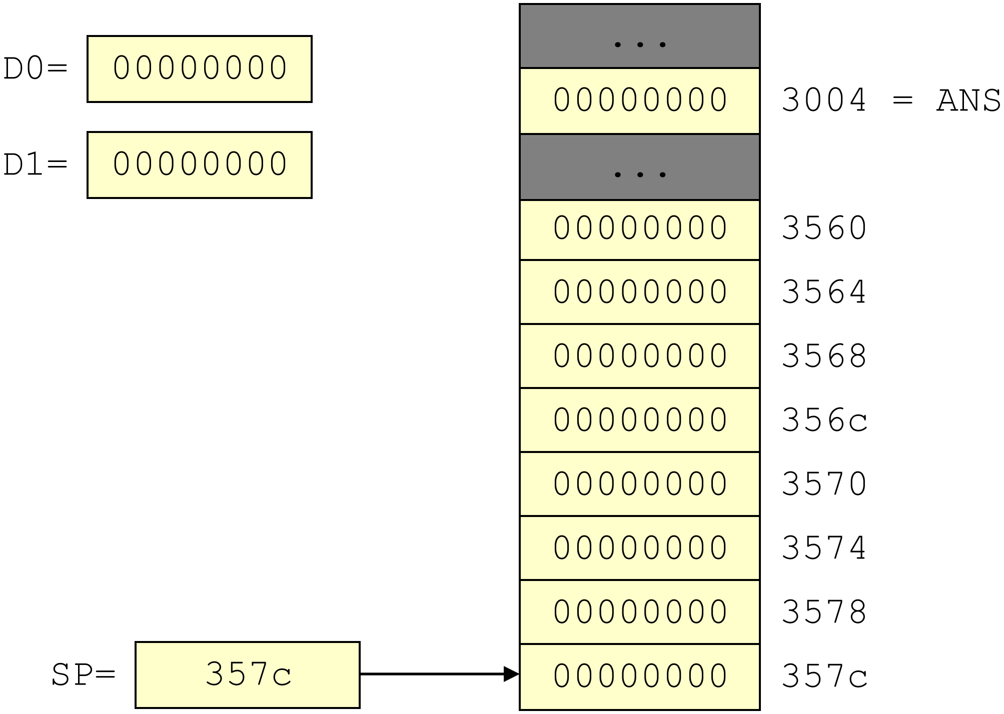
Restore n=3

ans = (3 * ans) = 3 *2

- Before calling yourself, you need to save copies of all your locally declared variables/parameters (e.g. n)

Recursive Routine Example

```
        .data
ANS:    .space   4

        .text = 0x20000500
START:  MOVE.L   #3,D0
        BSR.W    FACT
        MOVE.L   D1,ANS
        STOP     #$2700

        .org     0x300
FACT:   CMPI.L   #1,D0
        BEQ.S    NEQ1
        MOVE.L   D0,-(SP)
        SUBI.L   #1,D0
        BSR.S    FACT
        MOVE.L   (SP)+,D0
        MULU     D0,D1
        BRA.L    DONE
NEQ1:   MOVE.L   #1,D1
DONE:   RTS
```



D0= 00000000

D1= 00000000

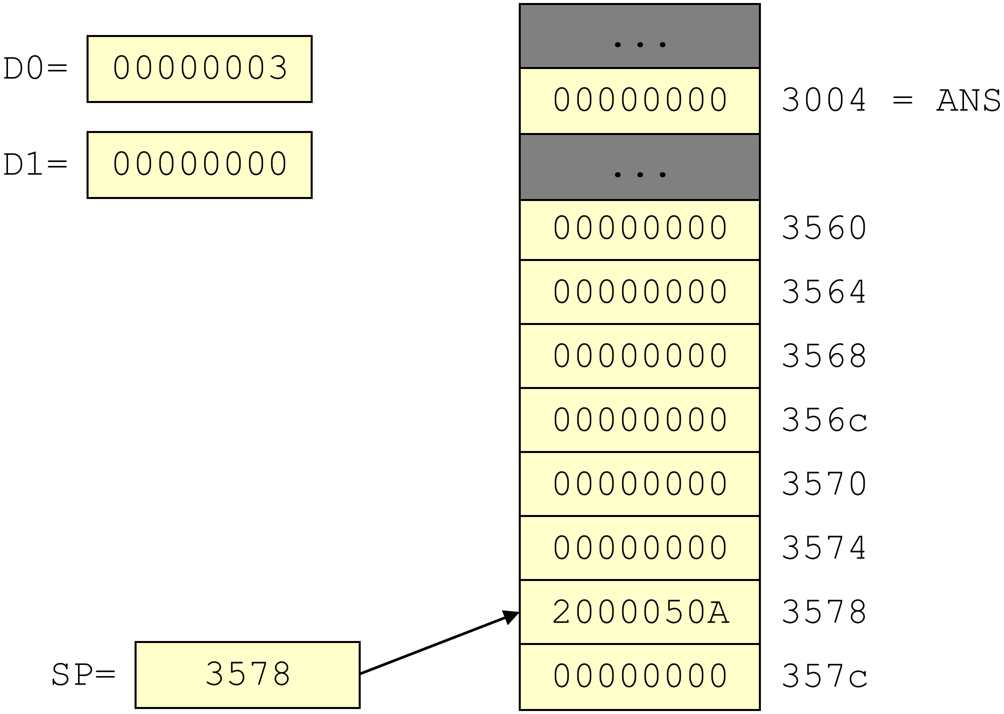
SP= 357c

. . .	
00000000	3004 = ANS
. . .	
00000000	3560
00000000	3564
00000000	3568
00000000	356c
00000000	3570
00000000	3574
00000000	3578
00000000	357c

- ANS is where we will place the final *n!* answer when finished calculating it. During calculation we will keep it in D1

Recursive Routine Example

```
            .data
ANS:        .space     4
            .text = 0x20000500
START:      MOVE.L     #3,D0
            BSR.W      FACT
0x050A      MOVE.L     D1,ANS
            STOP       #$2700
            .org       0x300
FACT:       CMPI.L     #1,D0
            BEQ.S      NEQ1
            MOVE.L     D0,-(SP)
            SUBI.L     #1,D0
            BSR.S      FACT
            MOVE.L     (SP)+,D0
            MULU       D0,D1
            BRA.L      DONE
NEQ1:       MOVE.L     #1,D1
DONE:       RTS
```



D0= 00000003

D1= 00000000

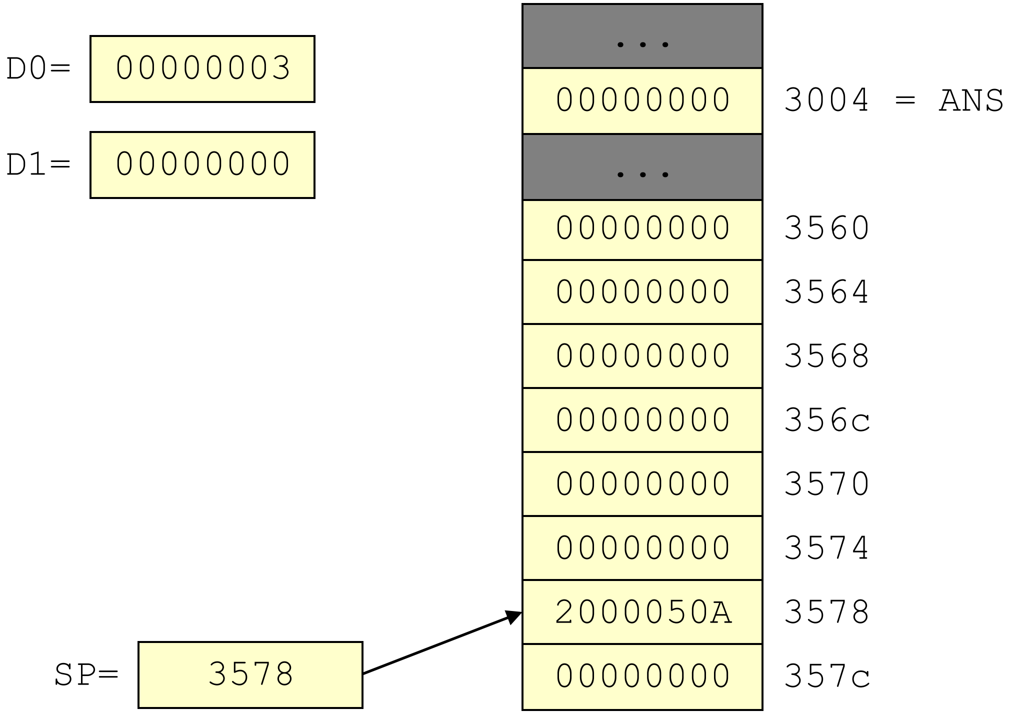
SP= 3578

. . .	
00000000	3004 = ANS
. . .	
00000000	3560
00000000	3564
00000000	3568
00000000	356c
00000000	3570
00000000	3574
2000050A	3578
00000000	357c

- Initialize N (D0) = 3
- Call Fact(3) => BSR pushes return address and goes to FACT

Recursive Routine Example

```
            .data
ANS:        .space     4
            .text = 0x20000500
START:      MOVE.L     #3,D0
            BSR.W      FACT
0x050A      MOVE.L     D1,ANS
            STOP       #$2700
            .org       0x300
FACT:       CMPI.L     #1,D0
            BEQ.S      NEQ1
            MOVE.L     D0,-(SP)
            SUBI.L     #1,D0
            BSR.S      FACT
            MOVE.L     (SP)+,D0
            MULU       D0,D1
            BRA.L      DONE
NEQ1:       MOVE.L     #1,D1
DONE:       RTS
```



D0= 00000003

D1= 00000000

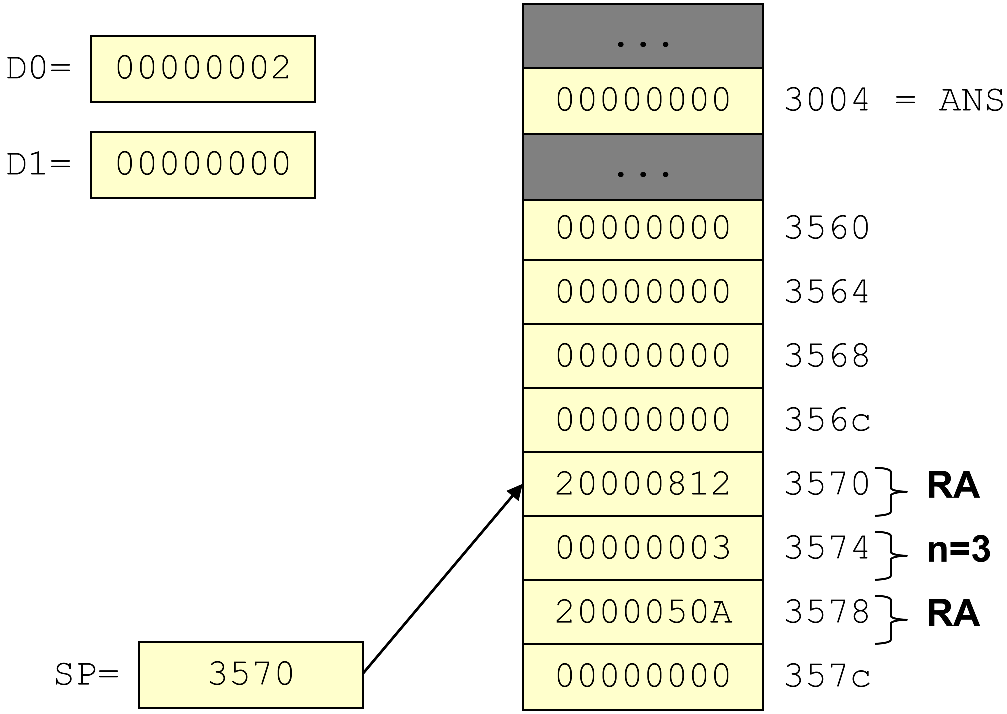
SP= 3578

. . .	
00000000	3004 = ANS
. . .	
00000000	3560
00000000	3564
00000000	3568
00000000	356c
00000000	3570
00000000	3574
2000050A	3578
00000000	357c

- Start by checking if N==1
- If not we continue sequentially

Recursive Routine Example

```
            .data
ANS:        .space    4
            .text = 0x20000500
START:      MOVE.L    #3,D0
            BSR.W     FACT
0x050A      MOVE.L    D1,ANS
            STOP      #$2700

            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
0x0812      MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```



D0= 00000002

D1= 00000000

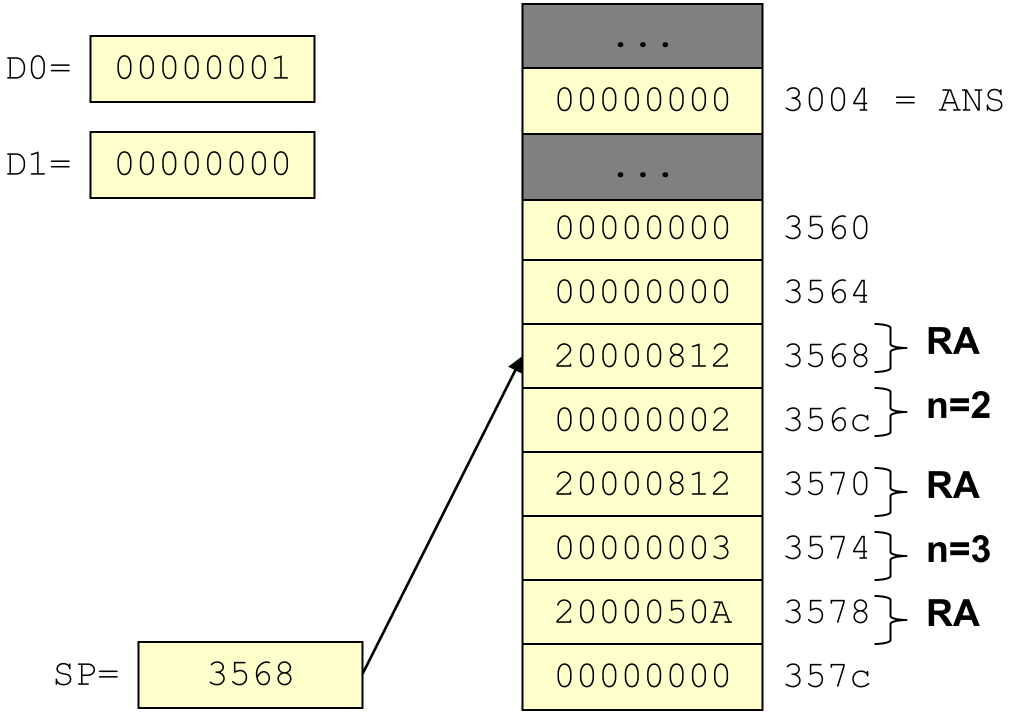
SP= 3570

...		
00000000	3004 = ANS	
...		
00000000	3560	
00000000	3564	
00000000	3568	
00000000	356c	
20000812	3570	RA
00000003	3574	n=3
2000050A	3578	RA
00000000	357c	

- We want to call Fact(2) but that means changing n (i.e. D0) to 2. We don’t want to lose our current value of 3 so we push it on the stack first.
- Then we decrement n and call Fact(2) pushing the RA

Recursive Routine Example

```
            .data
ANS:        .space    4
            .text = 0x20000500
START:      MOVE.L    #3,D0
            BSR.W     FACT
0x050A      MOVE.L    D1,ANS
            STOP      #$2700
            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
0x0812      MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```



D0= 00000001

D1= 00000000

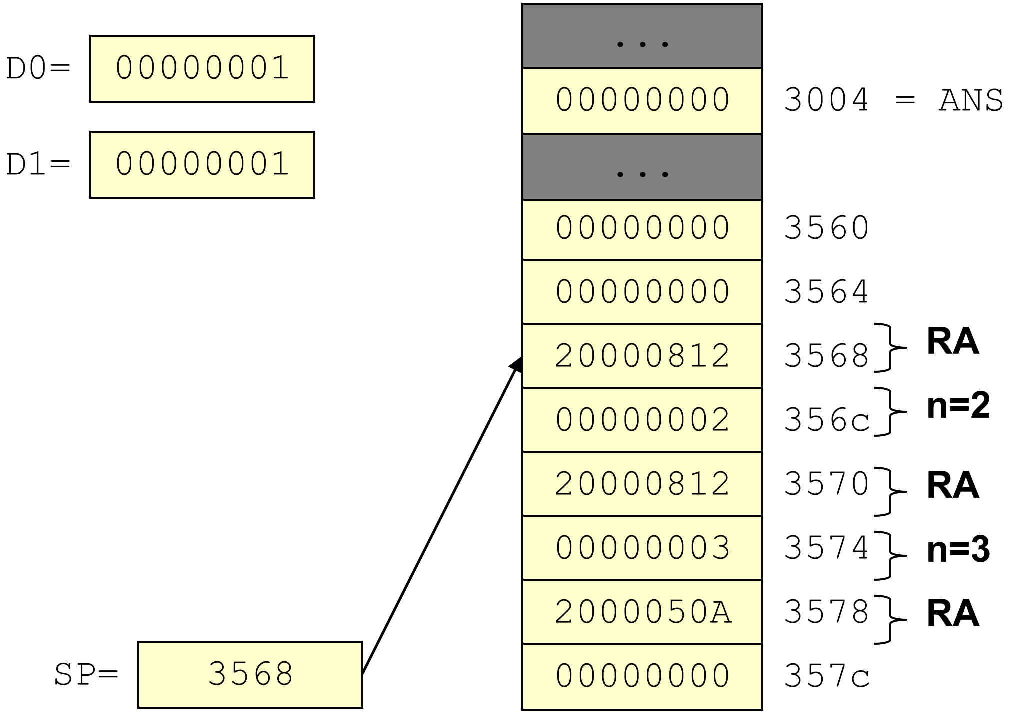
SP= 3568

. . .		
00000000	3004 = ANS	
. . .		
00000000	3560	
00000000	3564	
20000812	3568	RA
00000002	356c	n=2
20000812	3570	RA
00000003	3574	n=3
2000050A	3578	RA
00000000	357c	

- We now perform the check of n again, find that it is not equal to 1, save n=2 and decrement n=1 in order to call Fact(1)

Recursive Routine Example

```
            .data
ANS:        .space    4
            .text = 0x20000500
START:      MOVE.L    #3,D0
            BSR.W     FACT
0x050A      MOVE.L    D1,ANS
            STOP      #$2700
            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
0x0812      MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```



D0= 00000001

D1= 00000001

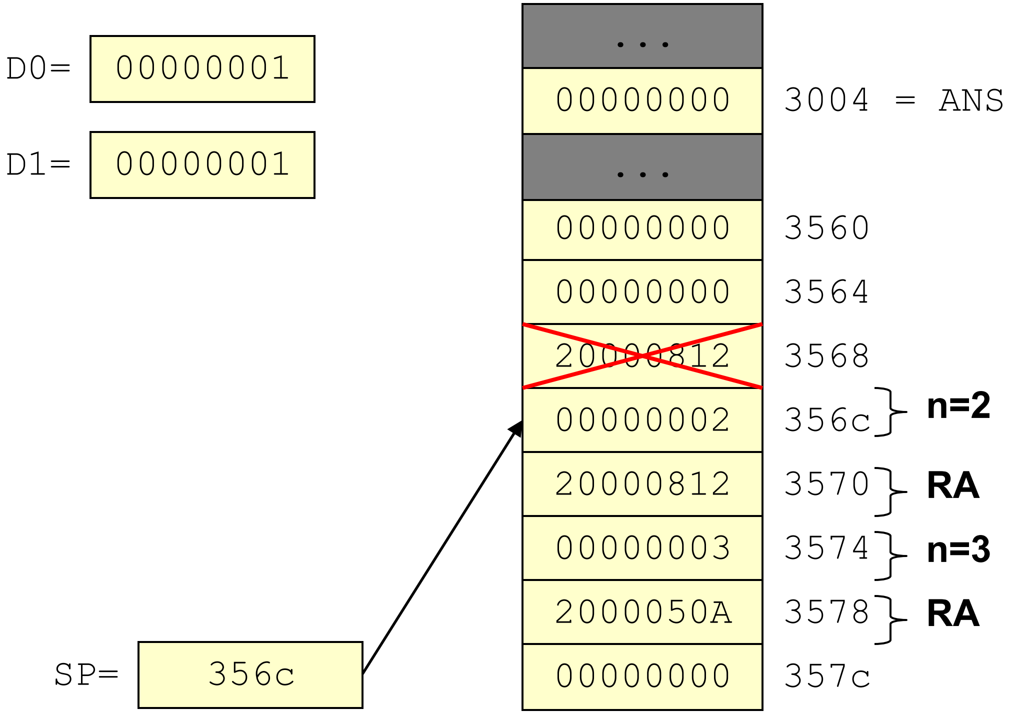
SP= 3568

. . .		
00000000	3004 = ANS	
. . .		
00000000	3560	
00000000	3564	
20000812	3568	RA
00000002	356c	n=2
20000812	3570	RA
00000003	3574	n=3
2000050A	3578	RA
00000000	357c	

- In Fact(1) our comparison will find that n is equal to 1, branch to NEQ1 and store the value 1 in D1 (i.e. ans)

Recursive Routine Example

```
                .data
ANS:            .space    4
                .text = 0x20000500
START:          MOVE.L    #3,D0
                BSR.W     FACT
0x050A          MOVE.L    D1,ANS
                STOP      #$2700
                .org      0x300
FACT:           CMPI.L    #1,D0
                BEQ.S     NEQ1
                MOVE.L    D0,-(SP)
                SUBI.L    #1,D0
                BSR.S     FACT
0x0812          MOVE.L    (SP)+,D0
                MULU      D0,D1
                BRA.L     DONE
NEQ1:           MOVE.L    #1,D1
DONE:           RTS
```



D0= 00000001

D1= 00000001

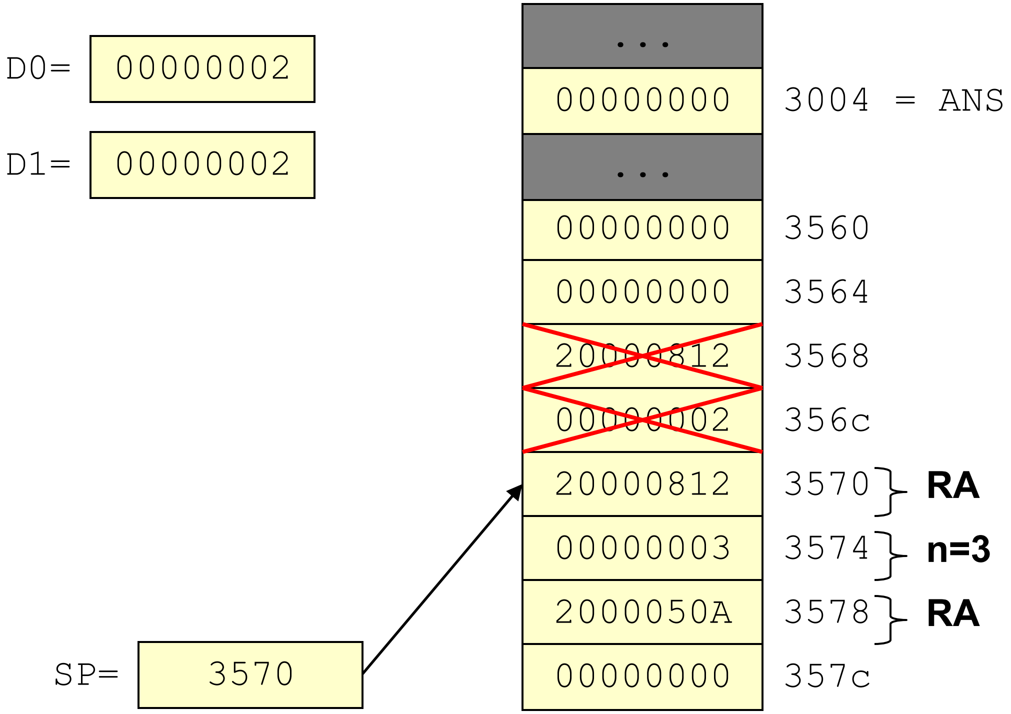
SP= 356c

. . .		
00000000	3004 = ANS	
. . .		
00000000	3560	
00000000	3564	
~~20000812~~	3568	
00000002	356c	n=2
20000812	3570	RA
00000003	3574	n=3
2000050A	3578	RA
00000000	357c	

- RTS will pop off the RA of $880E and go back to that instruction, effectively returning us into the context of Fact(2)

Recursive Routine Example

```
            .data
ANS:        .space    4
            .text = 0x20000500
START:      MOVE.L    #3,D0
            BSR.W     FACT
0x050A      MOVE.L    D1,ANS
            STOP      #$2700
            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
0x0812      MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```



D0= 00000002

D1= 00000002

SP= 3570

. . .

00000000 3004 = ANS

. . .

00000000 3560

00000000 3564

~~20000812~~ 3568

~~00000002~~ 356c

20000812 3570 } RA

00000003 3574 } n=3

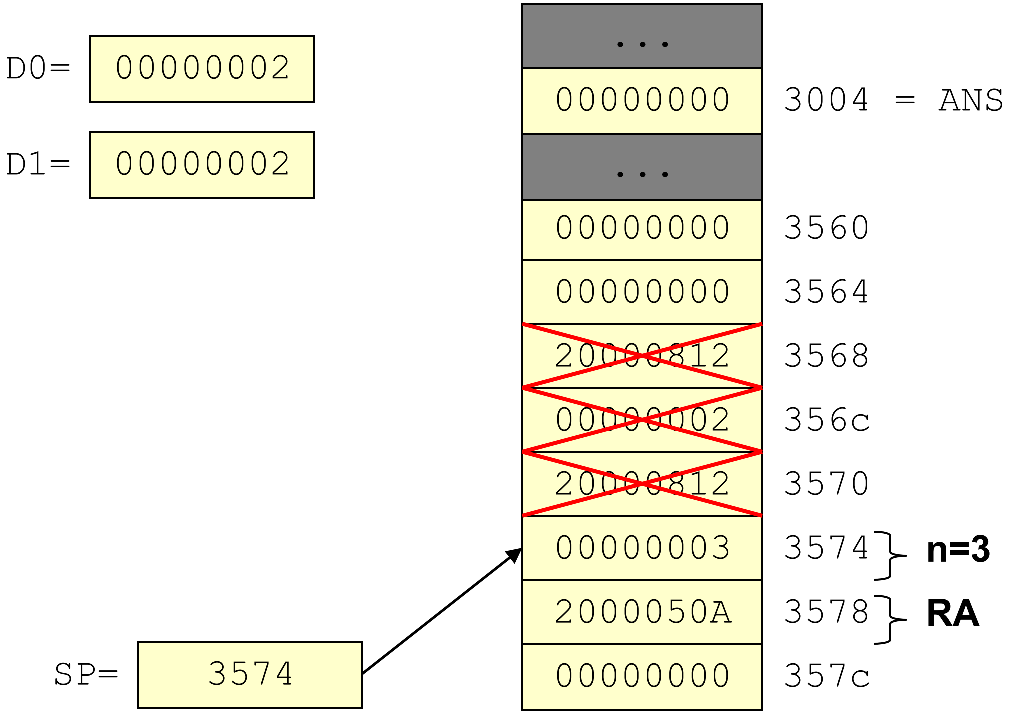
2000050A 3578 } RA

00000000 357c

- We will first restore our value of n = 2 by popping it off the stack back into D0
- We then calculate the factorial by taking ans = n * ans

Recursive Routine Example

```
           .data
ANS:       .space    4
           .text = 0x20000500
START:     MOVE.L    #3,D0
           BSR.W     FACT
0x050A     MOVE.L    D1,ANS
           STOP      #$2700
           .org      0x300
FACT:      CMPI.L    #1,D0
           BEQ.S     NEQ1
           MOVE.L    D0,-(SP)
           SUBI.L    #1,D0
           BSR.S     FACT
0x0812     MOVE.L    (SP)+,D0
           MULU      D0,D1
           BRA.L     DONE
NEQ1:      MOVE.L    #1,D1
DONE:      RTS
```



D0= 00000002

D1= 00000002

SP= 3574

...

00000000 3004 = ANS

...

00000000 3560

00000000 3564

~~20000812~~ 3568

~~00000002~~ 356c

~~20000812~~ 3570

00000003 3574 } n=3

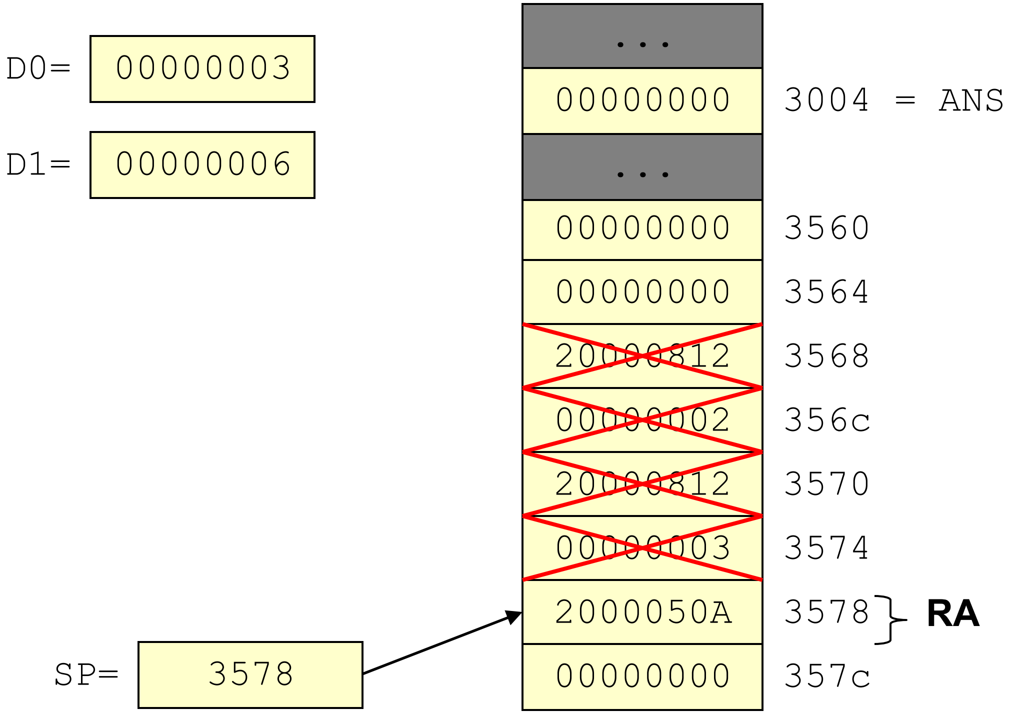
2000050A 3578 } RA

00000000 357c

- RTS is executed again, popping the RA off the stack and returning us into the context of Fact(3)

Recursive Routine Example

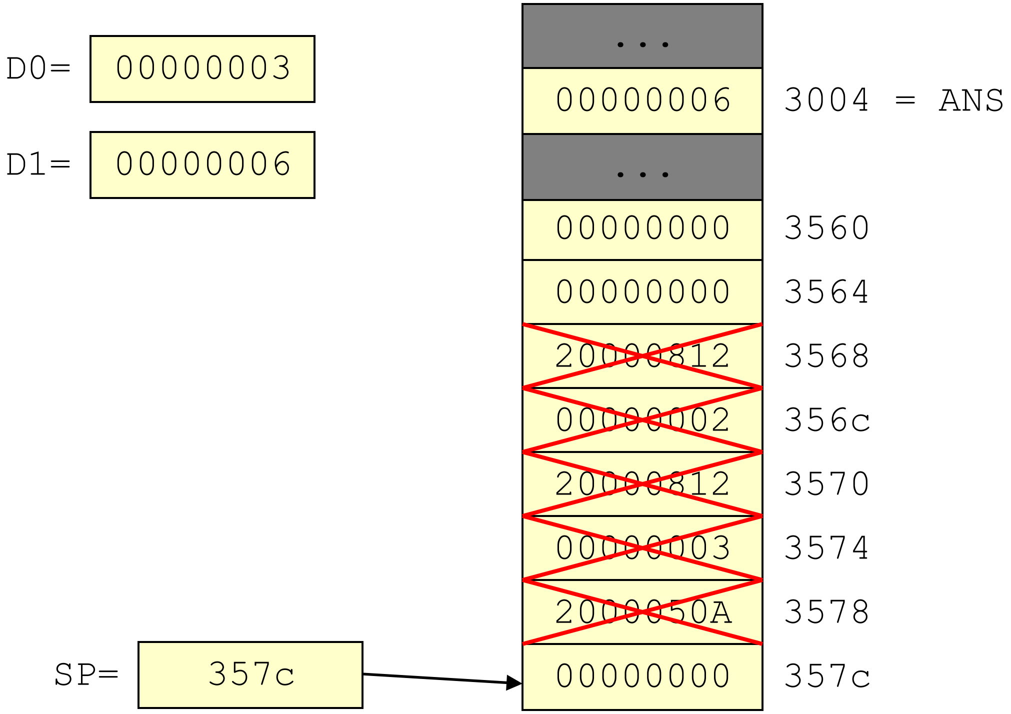
```
            .data
ANS:        .space    4
            .text = 0x20000500
START:      MOVE.L    #3,D0
            BSR.W     FACT
0x050A      MOVE.L    D1,ANS
            STOP      #$2700
            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
0x0812      MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```



- We again restore our value of n = 3 by popping it off the stack back into D0
- We then calculate the factorial by taking ans = n * ans

Recursive Routine Example

```
            .data
ANS:        .space    4

            .text = 0x20000500
START:      MOVE.L    #3,D0
            BSR.W     FACT
0x050A      MOVE.L    D1,ANS
            STOP      #$2700

            .org      0x300
FACT:       CMPI.L    #1,D0
            BEQ.S     NEQ1
            MOVE.L    D0,-(SP)
            SUBI.L    #1,D0
            BSR.S     FACT
0x0812      MOVE.L    (SP)+,D0
            MULU      D0,D1
            BRA.L     DONE
NEQ1:       MOVE.L    #1,D1
DONE:       RTS
```



D0= 00000003

D1= 00000006

SP= 357c

. . .	
00000006	3004 = ANS
. . .	
00000000	3560
00000000	3564
~~20000812~~	3568
~~00000002~~	356c
~~20000812~~	3570
~~00000003~~	3574
~~2000050A~~	3578
00000000	357c

- Executing the RTS this time pops the last RA from the stack and returns us to the original calling function
- We then store the result to memory