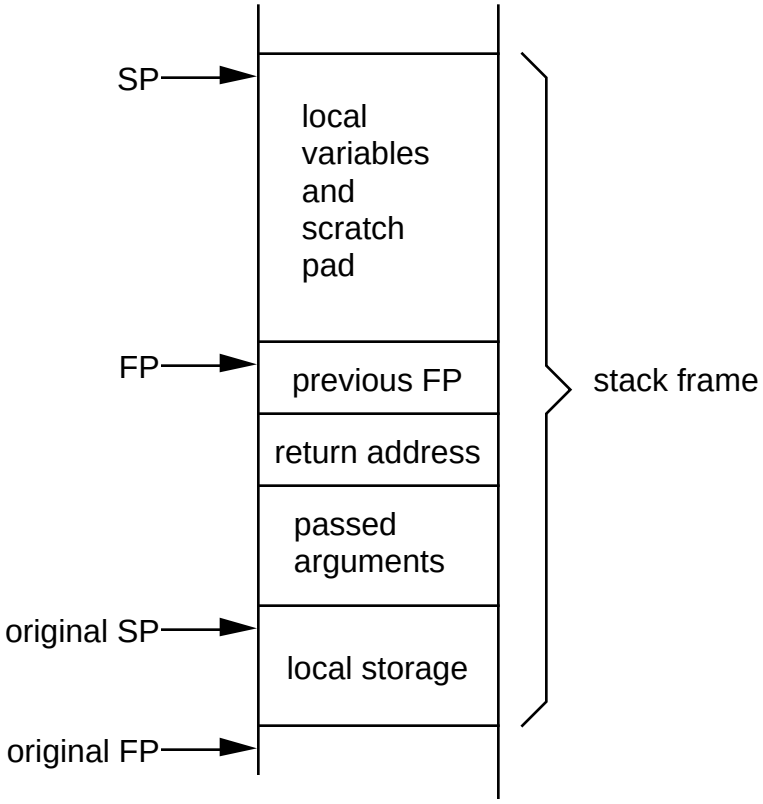
STACK FRAMES

The MC68000 provides two special instructions to allocate and deallocate a data structure called a frame in the stack to make subroutines easier to code.

general structure of a frame:



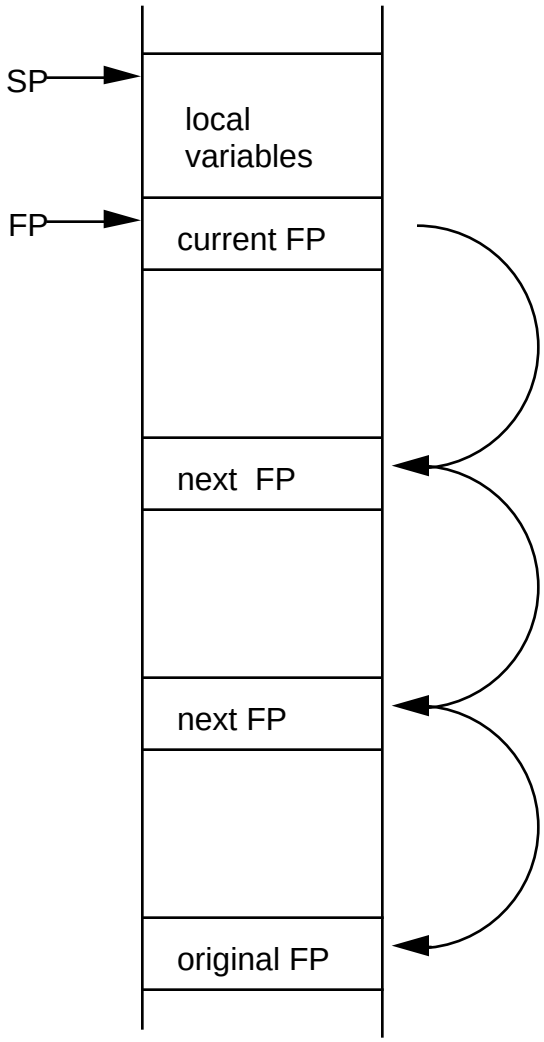
where register An is used as the argument pointer.

LINK An,d

1. put An at -(SP) Example: decrement stack pointer and put A0 on the stack.
2. put SP into An Example: set A0 to point to this value.
3. change SP-d to SP, i. e. decrement the SP

UNLK	An	1. An → SP, change the value of the SP to that contained in An 2. (SP)+ → An, put that value on the stack into An and deallocate that stack space.

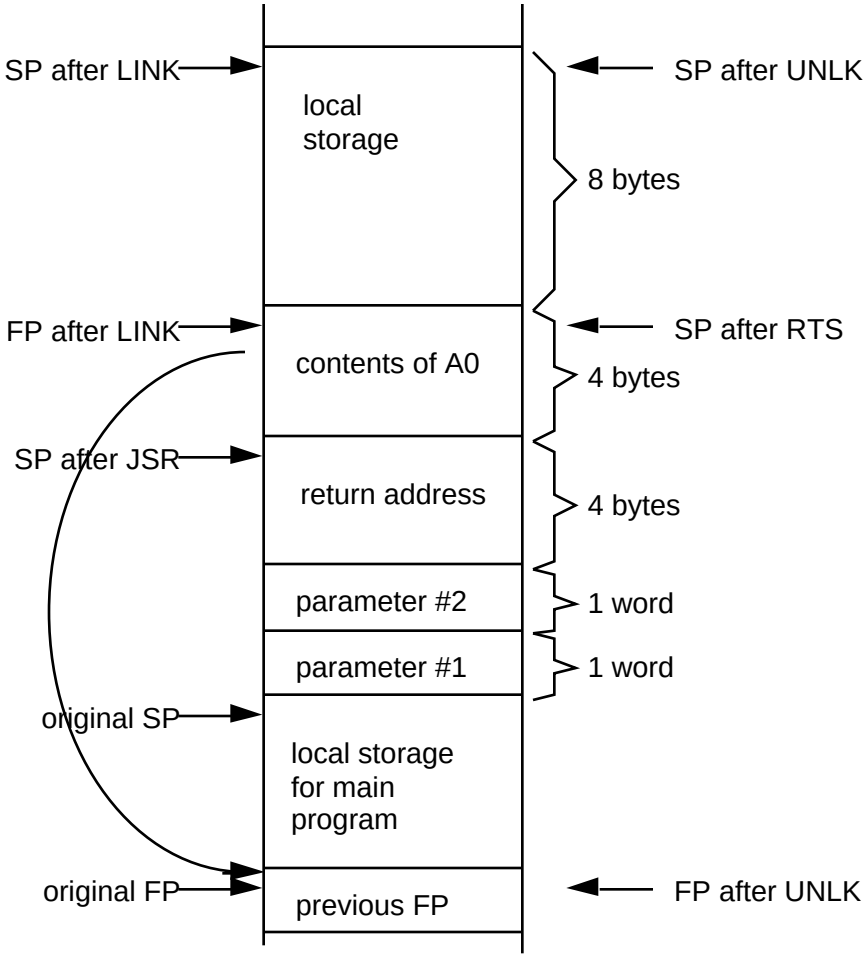
Return addresses and passed arguments are always positive relative to the frame pointer (FP).



Example:

```
        MOVE.W    D0,-(SP)      ;push parameter #1 onto stack
        MOVE.W    D1,-(SP)      ;push parameter #2 onto stack
        JSR       SBRT          ;jump to subroutine SBRT

SBRT    LINK      A0,-#$8       ;establish FP and local storage
        .
        .
        .
        MOVE.W    10(A0),D5     ;retrieve parameter #1
        .
        .
        .
        UNLK      A0            ;FP for the calling routine re-established.
                                Deallocate stack frame
        RTS                     ;return
```

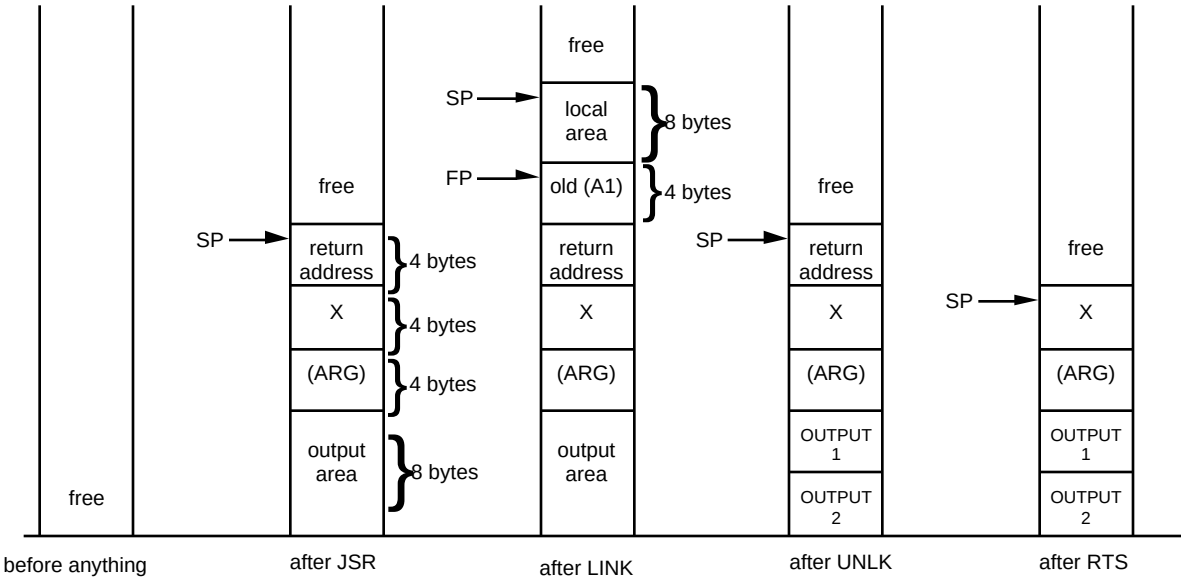
What the stack looks like during this program segment:



Note that the FP is stored in A0.

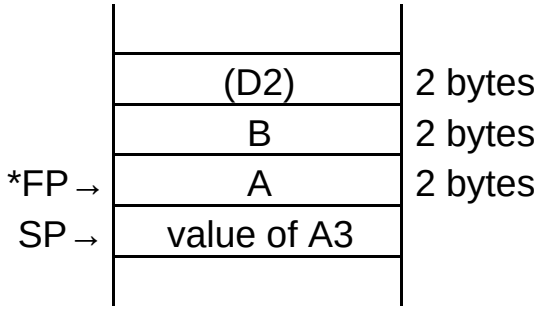
EXAMPLE:

```
ARG     DC.L                    ;number
N       EQU     8               ;8 bytes for output
M       EQU     8               ;8 bytes for local variables

        ADD.L   #-N,SP          ;put output area on stack
        MOVE.L  ARG,-(SP)       ;put argument on stack
        PEA     X               ;put address of data table
                                on stack
        JSR     SUBR            ;goto subroutine
        ADDA    #8,SP
        MOVE.L  (SP)+,D1        ;read outputs
        MOVE.L  (SP)+,D2
        •
        •
        •

SUBR    LINK    A1,#-M          ;save old SP
        •
        •
        •
        MOVE.L  LOCAL1,-4(A1)   ;save old variables
        MOVE.L  LOCAL2,-8(A1)   ;
        •
        •
        •
        ADD.L   #1,-4(A1)       ;change a local variable
        MOVEA.L 8(A1),A2        ;get X
        •
        •
        •
        MOVE.L  OUTPUT,16(A1)   ;push an output
        •
        •
        •
        UNLK    A1
        RTS

LOCAL1  DC.L    $98765432       ;local variables
LOCAL2  DC.L    $87654321
OUTPUT  DC.L    'ADCB'          output value
```



Program to compute the power of a number using a subroutine. Power MUST be an integer. A and B are signed numbers. Parameter passing using LINK and UNLK storage space on the stack.

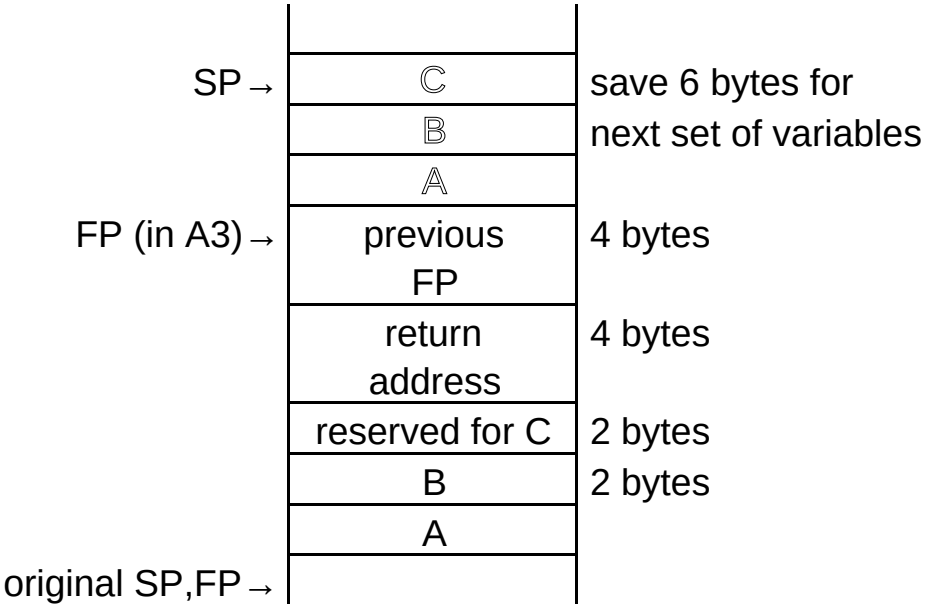
```
MAIN    LINK    A3,#-6          ;sets up SP
        MOVE    A,-2(A3)
        MOVE    B,-4(A3)
        JSR     POWR            ;call subroutine POWR
        LEA     C,A5
        MOVE    -6(A3),(A5)
        UNLK    A3

ARG     EQU     *
A       DC.W    4
B       DC.W    2
C       DS.W    1

POWR    EQU     *
        MOVE    -2(A3),D1       ;put A into D1
        MOVE    -4(A3),D2       ;put B into D2
        MOVE.L  #1,D3           ;put starting 1 into D3
LOOP    EQU     *
        SUBQ    #1,D2           ;decrement power
        BMI     EXIT            ;if D2-1<0 then quit  NOTE: this
                                gives us A**0=1
        MULS    D1,D3           ;multiply out power
        BRA     LOOP            ;and repeat as necessary
EXIT    EQU     *
        MOVE    D2,-6(A3)       ;C=(D3)
        RTS

        END     MAIN
```



*fixed while the SP changes

Better way.

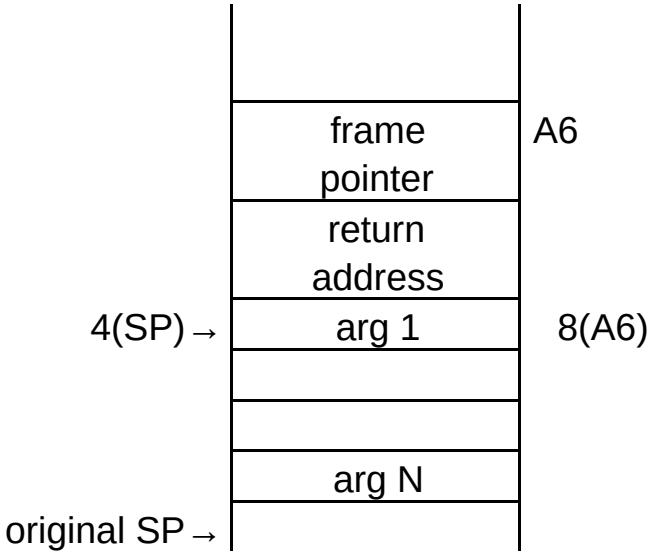
```
MAIN    MOVEA.L   SP,A3
        MOVE      A,-(SP)
        MOVE      B,-(SP)
        ADD.L     #2,SP          ;save output area
        JSR       POWR           ;call subroutine POWR
        LEA       C,A5
        MOVE      -6(A3), (A5)   ;put answer somewhere

ARG     EQU       *
A       DC.W      4
B       DC.W      2
C       DS.W      1

POWR    EQU       *
        LINK      A3,#-6
        MOVE      10(A3),D1      ;put A into D1
        MOVE      12(A3),D2      ;put B into D2
        •
        •
        •
        MOVE      D2,8(A3)       ;C=(D3)
        UNLK      A3
        RTS

        END       MAIN
```



Calling conventions for C or Pascal

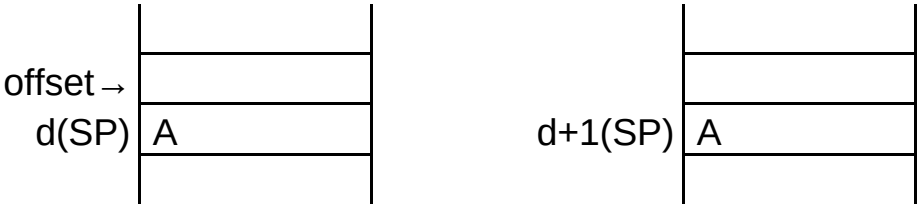
Arguments are pushed onto the stack in the reverse order of their appearance in the parameter list.

Just after a subroutine call:



If the function begins with a
LINK A6,#
High level language always generates LINK A6,# instructions

All arguments occupying just a byte in C are converted to a word and put in the low byte of the word, i.e.



Result, if any, is returned in D0 for function calls.

IT IS THE PROGRAMMER'S RESPONSIBILITY TO REMOVE THE ARGUMENTS FROM THE STACK.

The C calling sequence looks like this:

```
MOVE    ___,-(SP)   ;last argument
•
•
•
MOVE    ___,-(SP)   ;first argument
JSR     FUNCT
ADD     #N,SP       ;total size of arguments
```

Subroutine functions:

```
LINK    A6,#N
•
•
•
MOVE    ...,D0
UNLK    A6
RTS
```

The Pascal calling sequence pushes arguments in left to right order, then calls the function. The result if any is left on the stack. An example looks like this:

```
SUB     #N,SP       ;save space for result
MOVE    ...,-(SP)   ;push first argument onto stack
•
•
•
MOVE    ...,-(SP)   ;last argument
JSR     FUNCT
MOVE    (SP)+,...   ;store result
```

Subroutine code:

```
LINK    A6,#N
•
<code>
•
UNLK    A6
MOVE    (SP)+,A0    ;return address
ADD     #N,SP       ;total size of arguments
MOVE    ...,(SP)    ;store return result
JMP     (A0)
```

Symbols defined in assembly routines with the DS directive and exported using XDEF and XREF can be accessed from C as external variables. Conversely, C global variables can be imported and accessed from assembly using the XREF directive.

Miscellaneous comments about subroutines.

Parameter passing via MOVEM (move multiple registers)

If you have a small assembly language program this instruction allows you to save the values of registers NOT used to pass parameters.

Example:

```
SUBRTN  EQU     *
        MOVEM   D0-D7/A0-A6,SAVBLOCK
        •
        •
        •
        MOVEM   SAVBLOCK,D0-D7/A0-A6
```

where SAVBLOCK is local memory. This is bad practice since SAVBLOCK can be overwritten by your program.

MOVEM has two forms

```
        MOVEM   register_list,<ea>
        MOVEM   <ea>,register_list
```

More common to save registers on stack

```
SUBRTN  EQU     *
        MOVEM   D0-D7/A0-A6,-(SP)
        •
        •
        •
        MOVEM   (SP)+,D0-D7/A0-A6
        RTS
```

MOVEM is often used for re-entrant (subroutines that can be interrupted and re-entered) procedures.

The MOVEM instruction always transfers contents to and from memory in a predetermined sequence, regardless of the order used to specify them in the instruction.

address register indirect with pre-decrement	transferred in the order A7→A0, then D7→D0
for all control modes and address register indirect with post-increment	transferred in reverse order D0→D7, then A0→A7

This allows you to easily build stacks and lists.

Six methods of passing parameters:

1. Put arguments in D0 thru D7 before JSR (good only for a few arguments)
2. Move the addresses of the arguments to A0-A6 before JSR
3. Put the arguments immediately after the call. The argument addresses can be computed from the return address on the stack.
4. Put the addresses of the arguments immediately after the call in the code.
5. The arguments are listed in an array. Pass the base address of the array to the subroutine via A0-A6.
6. Use LINK and UNLK instructions to create and destroy temporary storage on the stack.

JUMP TABLES

— are similar to CASE statements in Pascal
— used where the control path is dependent on the state of a specific condition

EXAMPLE:

This subroutine calls one of five user subroutines based upon a user id code in the low byte of data register D0. The subroutine effects the A0 and D0 registers.

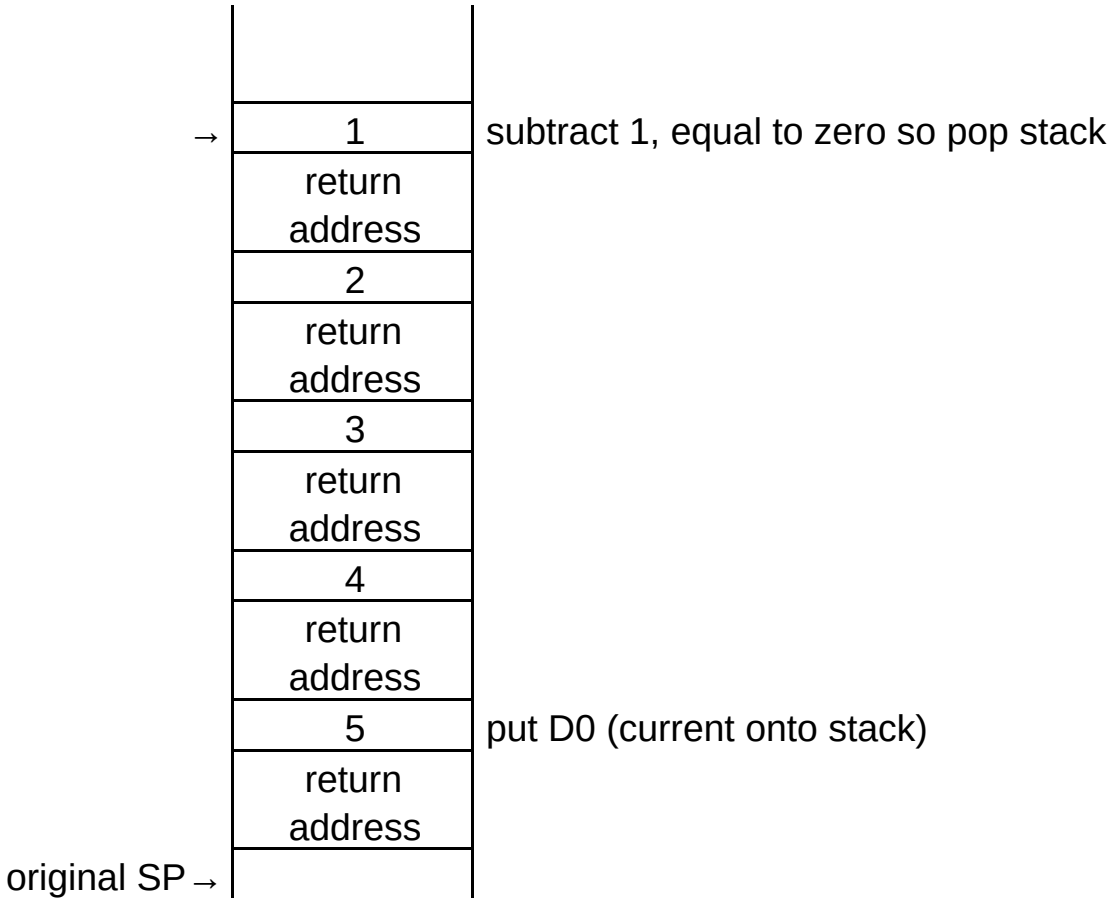
```
        RORG    $1000           ;causes relative addressing
                                (NOTE 1)
SELUSR  EXT.W   D0              ;extend user id code to word
        CHK     #4,D0           ;invalid id code ? (NOTE 2)
        LSL     #2,D0           ;NO! Calculate index=id*4
                                since all long word
                                addresses
        LEA     UADDR,A0        ;load table addresses
        MOVEA.L 0(A0,D0.W),A0   ;compute address of user
                                specified subroutine and put
                                correct caling address into
                                A0
        JMP     (A0)            ;jump to specified routine
        •
        •
        •
UADDR   DC.L    USER0,USER1,USER2,USER3,USER4
```

NOTES:

1. The RORG is often used when mixing assembly language programs with high level programs. It causes subsequenct addresses to be relative.
2. The CHK is a new instruction. In this case it checks if the least significant word of D0 is between 0 and 4 (2's complement). If the word is outside these limits, a exception through vector address $10 is initiated. The CHK instruction checks for addresses outside assigned limits and is often used to implement subscript checking.

EXAMPLE RECURSIVE PROCEDURE USING STACK

```
DATA    EQU     $6000
PROGRM  EQU     $4000

        ORG     DATA
NUMB    DS.W    1           ;number to be factorialized
F_NUMB  DS.W    1           ;factorial of input number

        ORG     PROGRM
MAIN    MOVE.W  NUMB,D0     ;get input number
        JSR     FACTOR      ;compute factorial
        MOVE.W  D0,F_NUMB   ;save the answer

* SUBROUTINE FACTOR
* PURPOSE: Determine the factorial of a given number.
* INPUT: D0.W  = number whose factorial is to be computed
*              0 ≤ D0.W ≤ 9
* OUTPUT: D0.W = factorial of input number
* REGISTER USAGE: No registers except D0 effected
* SAMPLE CASE: INPUT: D0.W=5
*              OUTPUT: D0.W=120

FACTOR  MOVE.W  D0,-(SP)    ;push current number onto
                            stack
        SUBQ.W  #1,D0       ;decrement number
        BNE.S   F_CONT      ;not end of factorial
                            computations
        MOVE.W  (SP)+,D0    ;factorial=1
        BRA.S   RETURN
F_CONT  JSR     FACTOR
        MULU    (SP)+,D0
RETURN  RTS
```

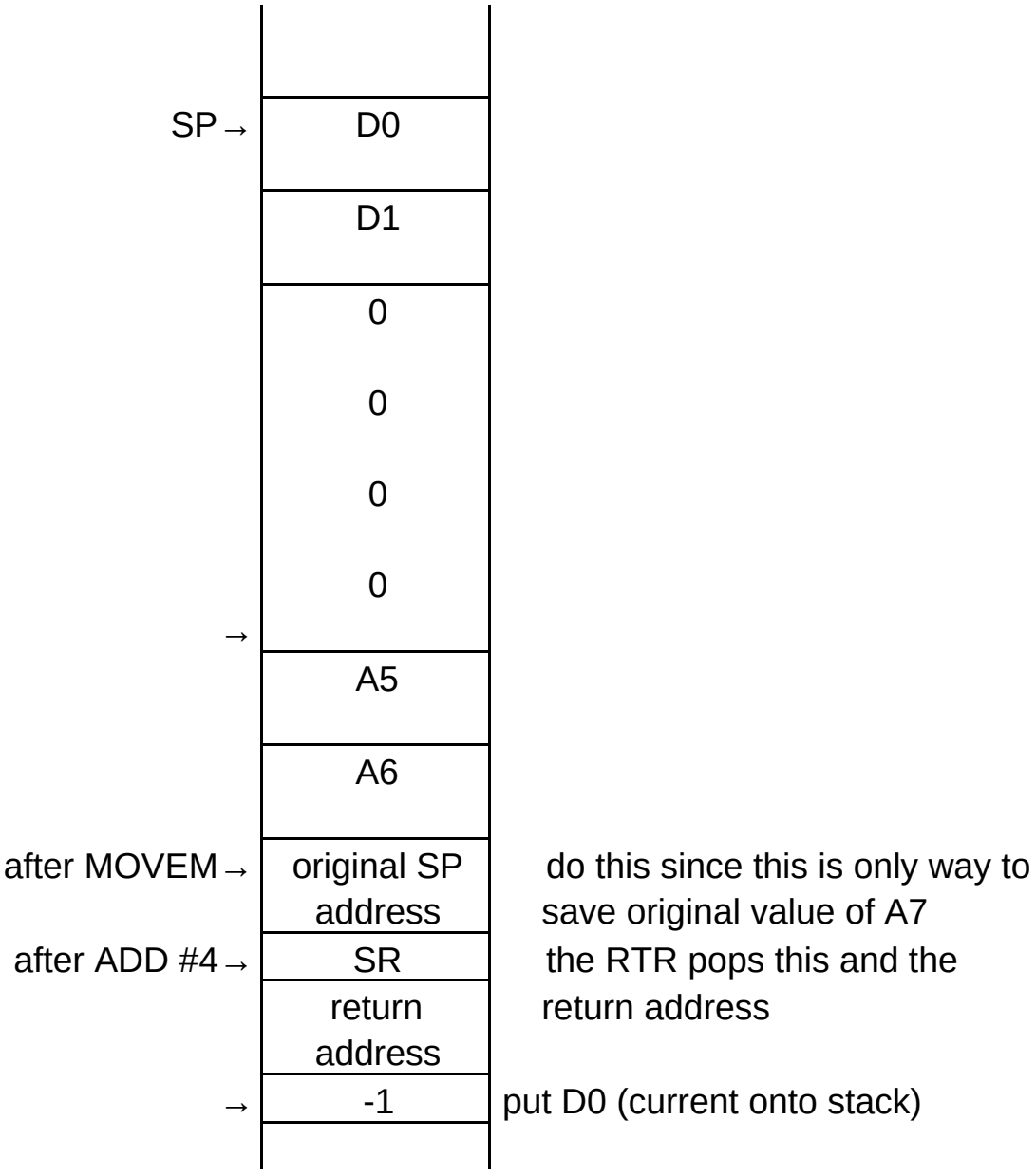


EXAMPLE
This is a simplified version of TUTOR's "DF" command. It uses the stack to display register contents.

```
START    MOVEM.L  TESTREGS,D0-D7/A0-A6    ;assign values to
                                            registers
         MOVE,L   #-1,-(SP)               ;put something on stack
         JSR      PRINTR                  ;print all registers
         MOVE.L   (SP)+,D0                ;retrieve it
         ADDQ.L   #1,D0                   ;null it
         JSR      PRINTR                  ;print them all again
         TRAP     #0                      ;stop program

SAVESP   EQU      60

PRINTR            ;data for PRINTREGS
RMSGS:
         DC.B     '  D0  D1  D2  D3  D4  D5',0
         DC.B     '  D6  D7  A0  A1  A2  A3',0
         DC.B     '  A4  A5  A6  SP  SR  PC',0

;                 |<---- 55 characters long ---->|
SPACES   DC.B     '  ',0                   ;2 blanks
CONBUF   DS.B     10
ENDLINE  DC.B     $0D,$0A,0
; data for program
CH       DS.B     1
         DS.W     1
TSTREG   DC.L     1,2,3,4,5,6,7,8,$A,$AA,$AAA
         DC.L     $AAAA,$AAAAA,$AAAAAA,$AAAAAAA
         END
```

```
PRINTR  MOVE.W  SR,-(SP)          ;save SR on stack
        PEA     6(SP)             ;save original SP on stack
        MOVEM.L D0-D7/A0-A6,-(SP) ;save all regular
                                  registers
        MOVEQ   #2,D4             ;D1 counts # of rows in
                                  printout
        MOVEA.L SP,A1             ;use A1 to point to beginning
                                  of data
        LEA     RMSGS,A2          ;use A2 to point to row
                                  headings
MLOOP                             ;output routine for heading
        MOVEA.L A2,A0             ;set pointer to beginning of
                                  header to be printed
        JSR     PrintString       ;output heading
        MOVEQ   #5,D5             ;output six registers this line
RLOOP   TST.W   D4                ;tests for SR to be printed
        BNE.S   NOT_SR            ;SR requires special routine
        CMP.W   #1,D5             ;as it is only word length
        BNE.S   NOT_SR            ;register
        LEA     SPACES,A0         ;load addresses of spaces
        JSR     PrintString       ;print spaces with no new
                                  line
        MOVE.W  (A1)+,D0          ;put SR word into D0
        JSR     PNT4HX            ;unimplemented routine to
                                  convert 4 hex digits in D0 to
                                  ascii code for printing
        JSR     PrintString       ;print hex contents
        LEA     SPACES,A0         ;load address of spaces
        JSR     PrintString       ;print them with no line feed
        BRA.S   ENDRPL
NOT_SR  MOVE.L  (A1)+,D0          ;put register contents into D0
        JSR     PNT8HX            ;unimplemented routine to
                                  convert 8 hex digits in D0 to
                                  ascii code for printing
```

```
ENDRPL  DBF     D5,PRLOOP       ;decrement register counter,
                                started at 5
        LEA     ENDLINE,A0      ;print CR+LF
        JSR     PrintString
        ADDA.L  #55,A2          ;increment  heading pointer
        DBF     D4,MLOOP        ;goto another line
        MOVEM.L (SP)+,D0-D7/A0-A6
        ADDQ.W  #4,SP           ;skip over A7 to point to SR
        RTR                     ;return and restore registers
```



SYSTEMS PROGRAMMING

Covers input/output programming, exception processing, peripheral device interrupts

Chapter 12	6850 ACIA (Asynchronous Communications Interface Adapter), 68230 PIT (Programmable Interval Timer)
Chapter 13	Exception processing, i.e. service routines and single stepping
Chapter 14	Exception processing and interrupt processing, concurrent programming.

Interrupts and exceptions
Instructions that interrupt ordinary program execution to allow access to system utilities or when certain internally generated conditions (usually errors) occur.

Conditions interrupting ordinary program execution are called exceptions. Usually are caused by sources internal to the 68000. Interrupts are exceptions which are caused by sources external to the 68000.

Exceptions transfer control to the program controlling the system (usually a monitor program or an operating system).

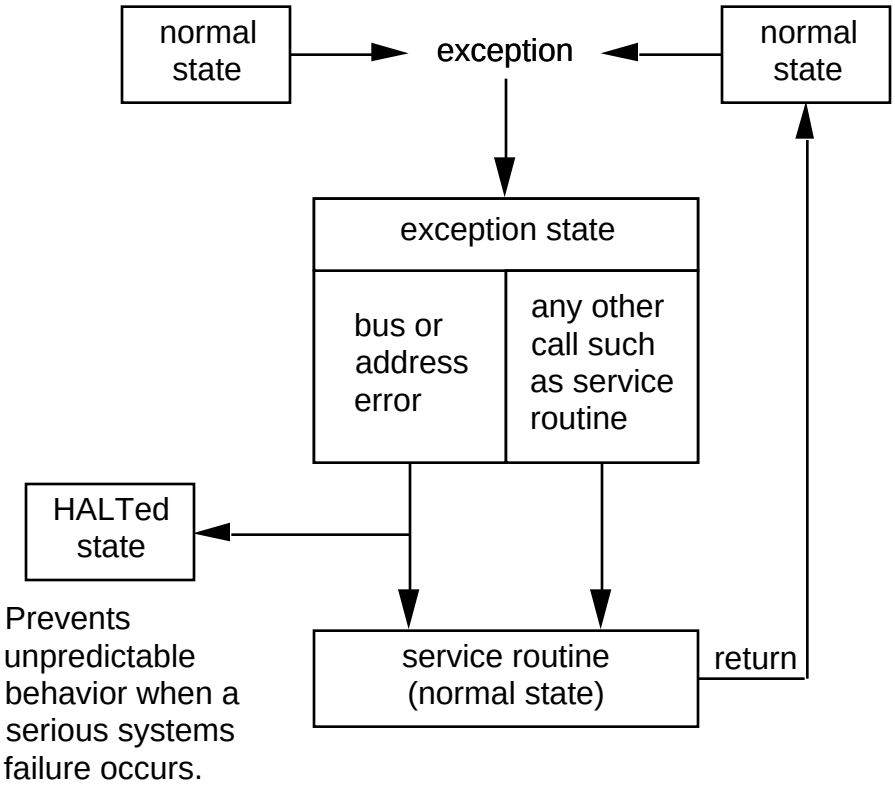
For example,

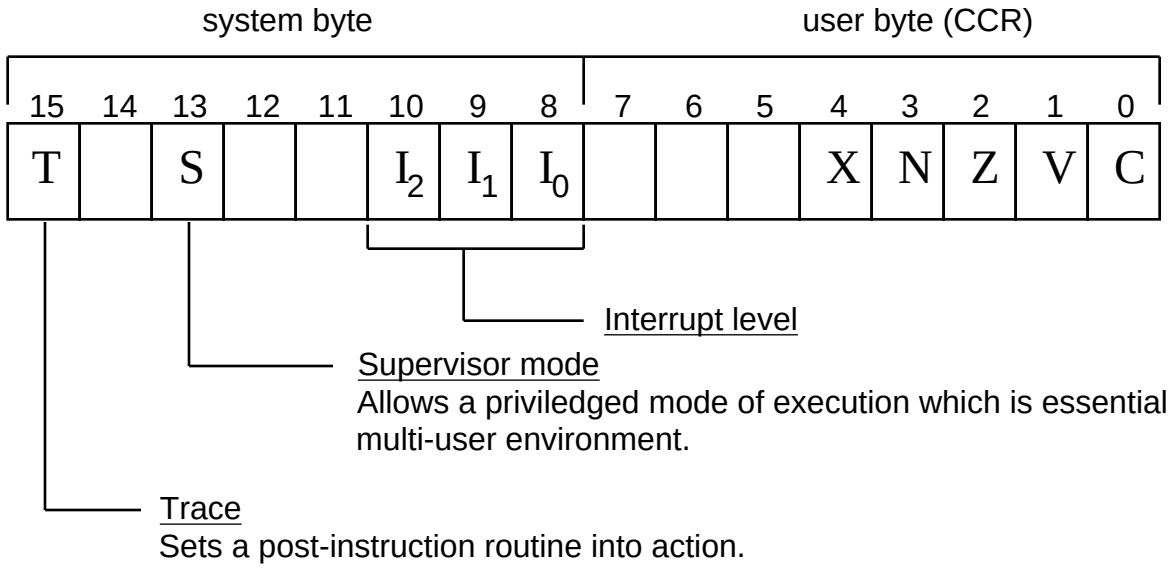
- user program executes a TRAP instruction for input/output which forces an exception.
- user program becomes suspended, file input/output is done by monitor program/operating system.
- user program is restarted where it was suspended.

As a result of any exception, the CPU switches from program execution to exception processing, services the exception request, and returns to normal program execution.

The MC68000 makes specific provision for two (actually three) operating states:

- normal state
- exception state
- HALTED state - used to prevent unpredictable behavior when a serious system failure occurs





Supervisor mode — single-user operating systems and monitor programs, all exception handling programs normally run in supervisor mode

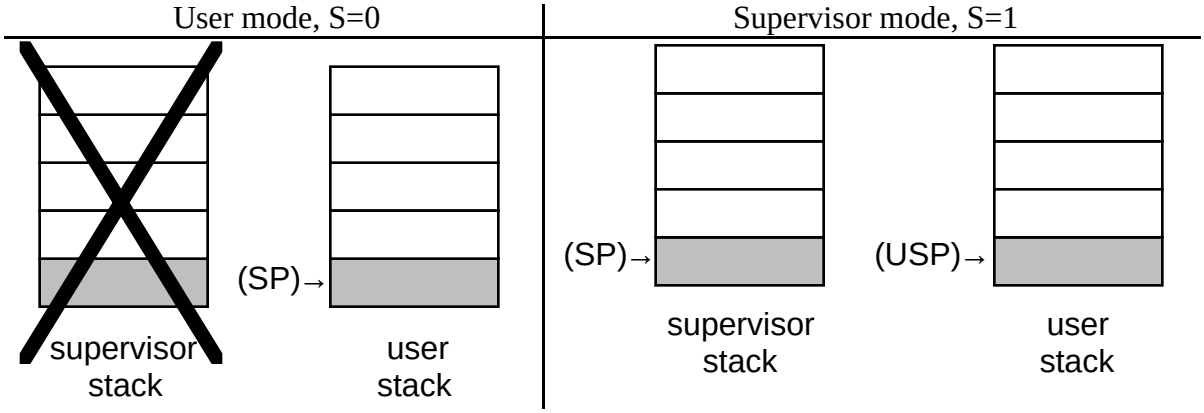
User mode — restricted access to the system environment, useful in multi-user environments.

The supervisor bit (bit 13 of the status register) is 1 if the 68000 is in supervisor (priviledged) mode.

There are four priviledged 68000 instructions; all have the entire SR as a destination.

```
MOVE.W      <ea>,SR
ANDI.W      #N,SR
ORI.W#N,SR
EORI.W      #N,SR
```

So that the 68000 does not become confused there are two stacks



Bit 13 of the status register is used to toggle A7 between the user and supervisor modes. USP references the user stack while the 68000 is in supervisor mode.

```
MOVE.L      USP,An
MOVE.L      An,USP
```

are the only instructions that can access the user stack while the 68000 is in supervisor mode. They are both priviledged instructions and transfer only long words (32 bits).

Examples 13.3

```
ADDA.L      D0,USP      ;not a valid instruction

MOVE.L      USP,A0      ;valid instruction, puts USP on system stack
MOVE.L      A0,-(SP)    ;cannot do MOVE.L USP,-(SP) directly
```

How exceptions are processed:

1.	identify the exception	• internal (identified by the CPU) - caused by TRAP instruction , etc. • external (identified by specific signal pins) - caused by hardware assertation
2.	save information about the currently running program	save the status register to a special internal register
3.	initialize the status register (for the exception routine)	set S=1, T=0 (typically), interrupt level for external exceptions
4.	determine the vector number	(from step 1), each exception type can have a unique routine. The MC68000 allows 255 such routines and stores their location (called a vector) addresses in the first 1K of 68000 program memory. This area is called the <u>exception vector table</u>. vector #1 SPECIAL for system start-up vector #2 • • • vector #255 address of exception vector = 4 × exception vector number
5.	save status register and return address	push the current PC (return address for the exception routine) and the <u>saved</u> status register onto the system stack.
6.	set PC to (address of exception vector), i.e. to correct starting address of exception service routine	the exception service routine is typically user created or part of your operating system. You are typically responsible for setting the vector address in the exception vector table.
7.	RTE	This is a priviledged insruction (Return from Exception) and MUST be at the end of each exception service routine. It pops the status register and PC from the supervisor stack. The status register and PC from the exception routine processing are lost.

Pseudo code exception processing cycle

identify exception vector number
save present status register to internal CPU register
set status register
    begin
    set S bit to 1
    set T bit to 0
    set interrupt level to level of present interrupt
    end
Compute exception vector address
* vector address = 4 × exception vector number
Save user information onto system stack
    begin
    push PC onto system stack
    push saved status register onto system stack
    end
Set PC to exception vector address

{Execute exception subroutine.}

RTE
* Similar to RTS, pops PC and SR from system stack,

EXCEPTION VECTOR TABLE

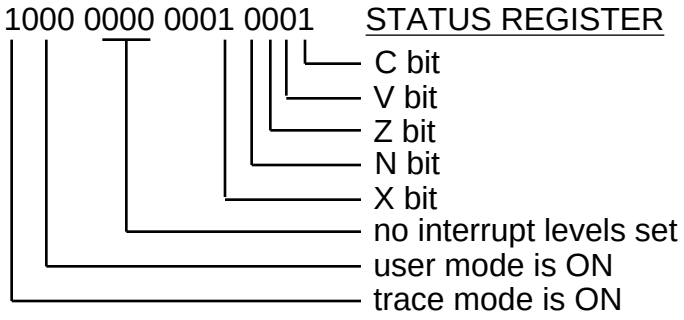
vector number (Decimal)	address (Hex)	assignment
0	0000	RESET: initial supervisor stack pointer (SSP)
1	0004	RESET: initial program counter (PC)
2	0008	bus error
3	000C	address error
4	0010	illegal instruction
5	0014	zero divide
6	0018	CHK instruction
7	001C	TRAPV instruction
8	0020	priviledge violation
9	0024	trace
10	0028	1010 instruction trap
11	002C	1111 instruction trap
12*	0030	not assigned, reserved by Motorola
13*	0034	not assigned, reserved by Motorola
14*	0038	not assigned, reserved by Motorola
15	003C	uninitialized interrupt vector
16-23*	0040-005F	not assigned, reserved by Motorola
24	0060	spurious interrupt
25	0064	Level 1 interrupt autovector
26	0068	Level 2 interrupt autovector
27	006C	Level 3 interrupt autovector
28	0070	Level 4 interrupt autovector
29	0074	Level 5 interrupt autovector
30	0078	Level 6 interrupt autovector
31	007C	Level 7 interrupt autovector
32-47	0080-00BF	TRAP instruction vectors**
48-63	00C0-00FF	not assigned, reserved by Motorola
64-255	0100-03FF	user interrupt vectors

NOTES:
* No peripheral devices should be assigned these numbers
** TRAP #N uses vector number 32+N

Example 13.4 TRACING

```
PC          instruction
            {somewhere in your program}
001FFC      MOVE.W      D0,16(A0)          ;trace service routine at $9000
002000      ADD.W       D2,D3

ORG         $9000
            {code for exception routine}
            RTE
```

The trace exception occurs after instruction is executed. For the purposes of this example, assume SR after MOVE.W is executed is $8011, i.e.



Consider pseudo code:

- Identifies exception vector number. For trace, exception vector number $=9_{10}$.
- Saves user SR to internal register.
- Sets new SR (upper bits only 0010 0000 = $20)
- Computes exception vector address ($9 \times 4 = 36_{10}$ = $24)
- Push user PC, then user SR onto system stack.
- Set PC = the contents of the exception vector address = ($24) = $9000
- Executes the trace routine which prints a message to the screen or printer and then clears the T bit in the user SR presently on the system stack.
- RTE

immediately after the MOVE instruction is executed	just before executing the interrupt service routine	after the RTE
SR: $8011* PC: $002000 The T bit is set indicating that TRACEing is in effect.	SR: $2011 PC: $009000 In special internal register: $8011 Notice that lower bits of SR are not altered by entering ISR.	SR: $0011* PC: $002000 * Recall that the exception routine turned off the T bit.
STACK: $8FFA $8FFC $8FFE (SP) → $9000	STACK: (SP) → **$8FFA** $8011 * $8FFC $0000 $8FFE $2000 * * $9000 * original SR ** original PC	STACK: $8FFA $8FFC $8FFE (SP) → $9000

HOW DOES THE 68000 START UP?

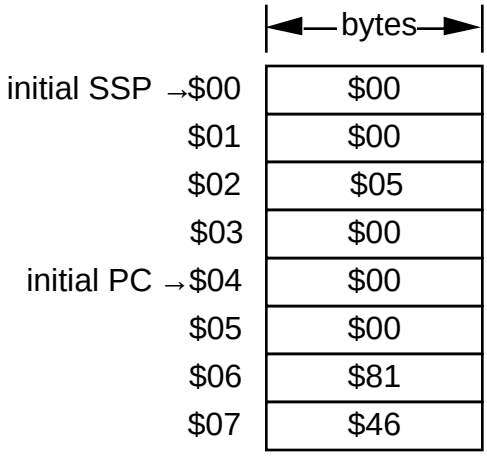
Hardware sets the RESET input pin (This is caused by circuitry external to the 68000). This causes a hardware exception which triggers the “reset exception.”

identifies the exception	RESET (#0)
no currently running program	
initialize the SR	S=1 T=0 I2I1I0 = 1112 (interrupts disabled)
determine vector number	0 in this case
save the SR and return address on the system stack	None, so the return stack must be initialized: (supervisor) SP = ($0) reset vector = [($0) ($4)]
set PC to (address of exception vector)	PC = ($4)

Example 13.5

Press RESET.	
Initialize SR.	SR=$2700
Initialize (SP).	(SP)=$0500
Initialize (PC)	(PC)=$8146
Begin execution of the 68000’s initialization routine at the starting address in memory.	

The memory (i.e. the RESET vector) in this example looks like:



Single board computers do NOT typically have an operating system. They have a simple program called a MONITOR which contains exception service routines whose starting addresses are loaded into the exception vector table at memory locations $8 - $3FF (remember the RESET vector MUST be in the first eight memory locations). Typically, the monitor will service key exceptions such as bus address errors, divide by zero, etc. with specific service routines. All other exceptions are handled by a generic service routine.

The startup sequence is special and consists of the following:

step	action	description
1.	set the status register to $2700	sets supervisor bit to 1, turns trace off, sets interrupt mask to 111
2.	set the Supervisor Stack Pointer to the contents of $0	SSP ← ($0.L)
3.	set the pc to the contents of $4	pc ← ($4.L)
4.	start program execution	

Example 13.4.2 SYSTEM INITIALIZATION

Address	exception	name of service routine
$08	bus error	VBUSERR
$0C	address error	VADDRERR
$10	illegal instruction	VILLEGINST
$14	divide by zero	VZERODIV
$18		VCHK
$1C		VTRAPV
$20	priviledged instruction violation	VPRIVINST
$24	trace (single step)	VTRACE
	generic routine	XHANDLE

Typical MONITOR routine:

```
* MONITOR INITIALIZATION ROUTINE
* ASSUMES RESET VECTOR CONTAINS ADDRESS OF INIT AT $4
STARTSP     EQU     $8000                   ;initial stack pointer
                                            value

* EXCEPTION VECTOR ADDRESSES IN SEQUENTIAL ORDER
VBUSERR     EQU     $08
VADDERR     EQU     $0C
VILLEGINST  EQU     $10
VZERODIV    EQU     $14
VCHK        EQU     $18
VTRAPV      EQU     $1C
VPRIVINST   EQU     $20
VTRACE      EQU     $24

* STORE EXCEPTION VECTORS IN THE ADDRESS TABLE

* RESET vector starts here
            ORG     $5000
INIT        LEA     STARTSP,SP              ;initialize SSP
            MOVE.L  #BUSERR,VBUSERR         ;initialize exception
            MOVE.L  #ADDERR,VADDERR         ;vector table
            MOVE.L  #ILLINST,VILLINST
            MOVE.L  #XHANDLE,VZERODIV
            MOVE.L  #XHANDLE,VCHK
            MOVE.L  #XHANDLE,VTRAPV
            MOVE.L  #PRIVIOL,VPRIVINST
            MOVE.L  #TRACE,VTRACE

            LEA     $28,A0                  ;load rest of the
                                            exception table from
                                            address $28 to $3FC
                                            with starting address
                                            of routine XHANDLE

ABINIT      MOVE.L  #XHANDLE,(A0)+
            CMPA.L  #$400,A0
            BCS.S   TABINIT

MAIN        {This is the mini-operating system and is a program that always runs.
            It might interpret commands, etc.)
            BRA     MAIN




* EXCEPTION SERVICE ROUTINES

BUSERR      {put code for routine here}

ADDERR      {put code for routine here}
```

```
ILLINST     {put code for routine here}

PRIVIOL     {put code for routine here}

TRACE       {put code for routine here}

XHANDLE                                         ;prints error message
            MOVEQ   #0,D0                       ;clear D1
            LEA     EXCEPTMSG,A0                ;load location of
                                                message
            JSR     PUTSTRING                   ;print it
            MOVE.L  2(SP),D0                    ;get return address
                                                from system stack
            JSR     PUTHEX                      ;print it
            JSR     NEWLINE
* FLUSH THE RETURN ADDRESS AND SR FROM THE SYSTEM STACK
            ADDQ.W  #6,SP                       ;flush the stack
            BRA     MAIN                        ;return to monitor

EXCEPTMSG   DC.B    'UNEXPECTED EXCEPTION AT ',0
```

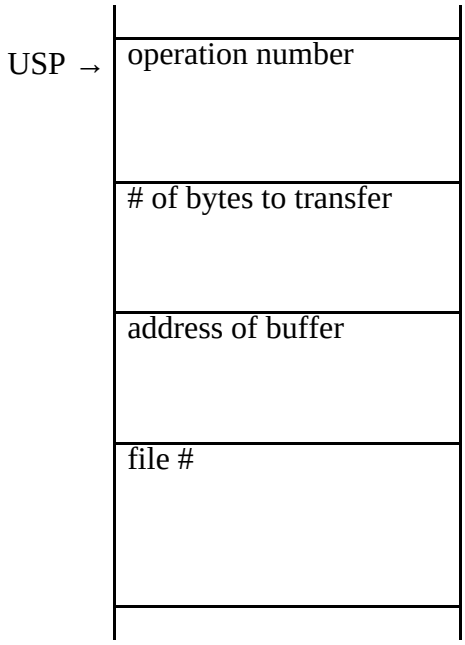
PROGRAM 13.1 TRAP HANDLER

```
VTRAP0      EQU     $80             ;trap #0 exception address, 128(10)
VTRAP1      EQU     $84             ;trap #1 exception address, 132(10)

STARTSP     EQU     $8000           ;initial SP value
STARTUSP    EQU     $4000           ;initial USP value
MONITOR     EQU     $8146           ;address of monitor

NULL        EQU     0
CONSOLE     EQU     0               ;console port

            XREF    INIT,PUTHEX,PUTSTRING,NEWLINE

* RESET VECTOR STARTS HERE
MAIN:       LEA     STARTSP,SP      ;initial stacks
            LEA     STARTUSP,A0
            MOVE.L  A0,USP

            MOVE.L  #TRAP0,VTRAP0   ;initialize exception vectors
            MOVE.L  #TRAP1,VTRAP1

            MOVEQ   #CONSOLE,D7     ;initialize UART specific to ECB
            JSR     INIT

* start program at address $2000 in user mode
            MOVE.L  #$2000,-(SP)    ;put starting address of user
                                    program on system stack
            MOVE.W  #$0000,-(SP)    ;clear status register
            RTE                     ;start user program

* service routine for TRAP #0.  Only a "file read" routine is simulated.
TRAP0:
            MOVEM.L D0/A0-A1,-(SP)
            MOVE.L  USP,A1          ;A1 points at user stack
            MOVE.L  (A1)+,D0        ;get the operation number
            ASL.L   #2,D0           ;4 byte index to iocalls table
                                    (multiply by 4 for byte offset)
            LEA     IOCALLS,A0      ;get base address of table
            MOVEA.L 0(A0,D0.L),A0   ;(A0) is address of the routine
* SUBROUTINE PUSHS ARGS OFF STACK
            JSR     (A0)            ;jump to routine
            MOVE.L  A1,USP          ;reset user stack pointer
            MOVEM.L (SP)+,D0/A0-A1
            RTE

* ROUTINE TO CREATE AND OPEN A  FILE ARE PLACED HERE
CREATE:
OPEN:

* READ ROUTINE DUMPS AND PRINTS PARAMETERS ON USER STACK
READ:       LEA     BYTEREAD,A0     ;get the bytes to read
            JSR     P[UTSTRING
```

```
                MOVE.L  (A1)+,D0
                JSR     PUTHEX
                JSR     NEWLINE
                LEA     BUFADDR,A0      ;get address of input buffer
                JSR     PUTSTRING
                MOVE.L  (A1)+,D0
                JSR     PUTHEX
                JSR     NEWLINE
                LEA     FILENUMBER,A0   ;get the system file number
                JSR     PUTSTRING
                MOVE.L  (A1)+,D0
                JSR     PUTHEX
                JSR     NEWLINE
                RTS

* ROUTINES TO WRITE TO A FILE AND CLOSE A FILE ARE PLACED HERE
WRITE:
CLOSE:

TRAP1:          JMP     MONITOR         ;perform the jump in supervisor
                                        mode
* DATA SECTION
IOCALLS         DC.L    CREATE,OPEN,READ,WRITE,CLOSE
BYTEREAD        DC.B    'BYTES TO READ:   ',NULL
BUFADDR         DC.B    'ADDRESS OF INPUT BUFFER:  ',NULL
FILENUMBER      DC.B    'FILE NUMBER:  ',NULL

                END
```

THE FOLLOWING PROGRAM IS ASSEMBLED AND LOADED AT ADDRESS $2000. PROGRAM 13.1 INITIATES EXECUTION OF THE PROGRAM AND PRODUCES THE OUTPUT BELOW:

```
                ORG         $2000
START:
* put input parameters on stack
                MOVE.L      #3,-(SP)        *FILE NUMBER IS 3
                MOVE.L      #BUF,-(SP)      *ADDRESS OF INPUT BUFFER
                MOVE.L      #512,-(SP)      *NUMBER OF BYTES TO READ
                MOVE.L      #2,-(SP)        *READ OPERATION IS 2
                TRAP        #0              *DO THE READ
                TRAP        #1              *TRAP #1 RETURNS CONTROL TO
                                            MONITOR WHEN IN USER MODE

BUF:            DS.B        512
                END
```



<RUN>

OUTPUT:
BYTES TO READ: 00000200
ADDRESS OF INPUT BUFFER: 0000201C
FILE NUMBER: 00000003

PROGRAM 13.2 ERROR HANDLER:

```
VDIV        EQU     $14                 ;divide by zero exception
                                        vector
VTRAPV      EQU     $1C                 ;trap on overflow exception
                                        address
MONITOR     EQU     $8146               ;address of monitor in ROM
NULL        EQU     0
RTNADDR     EQU     2                   ;offset to return address
STARTSP     EQU     $8000               ;starting SSP

            XREF    INIT,PUTSTRING,PUTHEX,NEWLINE

            ORG     $1000               ;program would be started by
                                        MONITOR program
START:      MOVE.L  #STARTSP,SP         ;initialize supervisor stack
            MOVE.L  #DIVERR,VDIV.W      ;initialize exception vectors
            MOVE.L  #OVRFLERR,VTRAPV.W

* INITIAL ACIA CONSOLE PORT
            MOVEQ   #0,D7
            JSR     INIT

* START UP A USER PROGRAM AT ADDRESS $2000
            MOVE.L  #$2000,-(SP)
            MOVE.W  #$2000,-(SP)
            RTE

* EXCEPTION SERVICE ROUTINES

DIVERR:     MOVEQ   #0,D7               ;the ACIA is the terminal
            LEA     DIVMSG,A0
            JSR     PUTSTRING
            MOVE.L  2(SP),D0            ;load return address into D0
            JSR     PUTHEX              ;print return address
            JSR     NEWLINE
            JMP     MONITOR             return user to MONITOR
* RTE IS NOT DONE HERE.  JUST ABORT THE PROGRAM
* PROBABLY SHOULD FLUSH STACK

OVRFLERR:   MOVEQ   #7,D1
            LEA     OVRFLMSG,A0
            JSR     PUTSTRING
            JSR     NEWLINE
            RTE

DIVMSG:     DC.B    'DIVIDE BY ZERO: PC =  ',NULL
OVRFLMSG:   DC.B    'OVERFLOW ',NULL

            ORG     $2000
START:      MOVE.W  #$5000,D0
            ADDI.W  #$4000,D0           ;V will be set
            TRAPV
```

```
                DIVS        #0,D0
                END

END
```

THE FOLLOWING PROGRAM IS ASSEMBLED AND LOADED AT ADDRESS $2000. PROGRAM 13.2 INITIATES EXECUTION OF THE PROGRAM AND PRODUCES THE OUTPUT BELOW:

```
<RUN>
OUTPUT:         OVERFLOW
                DIVIDE BY ZERO: PC =  0000200E
```

PROGRAM 13.3 SINGLE STEPPING

```
* CONTROL CHARACTERS
LINEFEED        EQU         $0A
NULL            EQU         $00

* EXCEPTION VECTORS
VTRACE                      EQU     $24
VBUSERROR                   EQU     $08
VADDRESSERROR               EQU     $0C
VILLEGALINSTRUCTION         EQU     $10
VPRIVILEDGEVIOLATION        EQU     $20
VTRAP0                      EQU     $80
VTRAP1                      EQU     $84

MONITOR         EQU         $8146

                XREF        INIT,PUTHEX,GETCHAR,NEWLINE
                XREF        ECHOFF,PUTSTRING,GETSTRING

* INITIALIZE THE SUPERVISOR AND USER STACK POINTERS
START:          LEA         $8000,SP
                LEA         $4000,A0
                MOVE.L      A0,USP

* INITIALIZE THE EXCEPTION VECTORS
                MOVE.L      #TRACE,VTRACE.W
                MOVE.L      #FATALERROR,VBUSERROR.W
                MOVE.L      #FATALERROR,VADDRESSERROR.W
                MOVE.L      #FATALERROR,VILLEGAL INSTRUCTION.W
                MOVE.L      #FATALERROR,VPRIVILEDGEVIOLATION.W
                MOVE.L      #OUTPUT,VTRAP0.W
                MOVE.L      #EXIT,VTRAP1.W

* INITIALIZE THE ACIA
                MOVEQ       #0,D7
                JSR         INIT
                JSR         ECHOFF

* START THE PROGRAM AT $3C00
                MOVE.L      #$3C00,-(SP)        ;starting address is $3C00
                MOVE.W      #$8000,-(SP)        ;start program in user mode,
                                                ;trace on, interrupt level 0
                RTE                             ;start the program

ENDRUN
                JMP         MONITOR             ;return to main monitor

* BUS/ADDRESS ERROR, ILLEGAL INSTRUCTION, PRIVILEDGE VIOLATION,
* TRAP SERVICE ROUTINES

FATALERROR:
                MOVEQ       #0,D7
                LEA         FATALMSG,A0
```

```
                JSR     PUTSTRING
                BRA     ENDRUN                  ;return to MONITOR

* USER PROGRAM EXECUTING TRAP #1 CAUSES TRAP TO HERE
EXIT:           BRA     ENDRUN                  ;return to MONITOR

* USER PROGRAM EXECUTING TRAP #0 CAUSES
* TRAP HERE FOR HEX OUTPUT
OUTPUT:         MOVE.L  D7,-(SP)
                MOVEQ   #0,D7
                JSR     NEWLINE
                JSR     PUTHEX                  ;output (D0)
                JSR     NEWLINE
                MOVE.L  (SP)+,D7
                RTE

* PRIMARY ROUTINE - CATCHES TRACE TRAP, DISPLAYS REGISTERS,
* AND HANDLES PROMPT FOR ANOTHER INSTRUCTION

PROGCOUNTER     EQU     2
TSIZE           EQU     6

TRACE:
                MOVEM.L D0-D7/A0-A6,GENREG.W
                MOVE.L  PROGCOUNTER(SP),D0      ;get PC
                MOVEQ   #0,D1                   ;get SR as a long word
                MOVE.W  (SP),D1
                LEA     TSIZE(SP),A0            ;original value of SP
                MOVE.L  USP,A1                  ;get USP
                MOVEM.L D0-D1/A0-A1,REGS.W      ;load PC/SR/SSP/USP to
                                                print
                MOVEQ   #0,D1                   ;output to console port
                MOVEQ   #1,D2                   ;allow four registers per line
                MOVEQ   #18,D3                  ;19 entries to print
                LEA     REGS.W,A1               ;saved registers begin at
                                                ADDRESS(SP)
                LEA     REGMSGS.W,A0
REGPL:          JSR     PUTSTRING
                MOVE.L  (A1)+,D0
                JSR     PUTHEX
                ADDA.L  #8,A0
                ADDQ.W  #1,D2                   ;count 4 registers per line
                CMPI.W  #4,D2
                BLE.S   NEXT
                MOVEQ   #1,D2
                JSR     NEWLINE
NEXT:           DBRA    D3,REGPL
                LEA     OPMSG,A0                ;print message about opcode
                                                word
                JSR     PUTSTRING
                MOVE.L  PROGCOUNTER(SP),A0      ;get opcode word of next
                                                instruction
                MOVEQ   #0,D0
                MOVE.W  (A0),D0
```

```
                JSR         PUTHEX                  ;print it
                LEA         PROMPT,A0               ;">>" prompt
                JSR         PUTSTRING
                JSR         GETCHAR                 ;read a character from
                                                    terminal
                CMPI.B      #LINEFEED,D0            ;carriage return, continue
                BEQ.S       RET
                ANDI.W      #$7FFF,(SP)             ;turn tracing off
RET:            JSR         NEWLINE
                MOVEM.L     GENREGS,D0-D7/A0-A6
                RTE                                 ;back to user program

REGS:
                DS.L        4                       ;to contain PC/SR/SSP/USP

GENREGS:        DS.L        15                      ;to contain D0-D7/A0-A6 at
                                                    each trace exception

REGMSGS:
        DC.B    '  PC = ',NULL,'  SR =  ',NULL,'  SSP= ',NULL
        DC.B    'USP = ',NULL,'  D0 =  ',NULL,'  D1= ',NULL
        DC.B    '  D2 = ',NULL,'  D3 =  ',NULL,'  D4= ',NULL
        DC.B    '  D5 = ',NULL,'  D6 =  ',NULL,'  D7= ',NULL
        DC.B    '  A0 = ',NULL,'  A1 =  ',NULL,'  A2= ',NULL
        DC.B    '  A3 = ',NULL,'  A4 =  ',NULL,'  A5= ',NULL
        DC.B    '  A6 = ',NULL'
OPMSG:
        DC.B    LINEFEED,'OPCODE WORD NEXT INSTRUCTION = 'NULL
PROMPT
        DC.B    LINEFEED,LINEFEED,'>> ',NULL
FATALMSG:
        DC.B    LINEFEED,'FATAL ERROR HAS OCCURRED',LINEFEED,NULL

                ORG         $3C00
START:          MOVE.L      #5,-(SP)
                MOVEQ       #5,D5
                LEA         OUTPUT(PC),A3
                MOVE.W      #$001F,CCR
OUTPUT:         MOVE.L      #$55553333,D0
                TRAP        #0
                TRAP        #1

                END
```

The actual program assembles to:

```
                ORG         $3C00
START:
003C00          2F3C 0000 0005          MOVE.L  #5,-(SP)
003C06          7A05                    MOVEQ   #5,D5
003C08          47FA 0006   LEA         OUTPUT(PC),A3
003C0C          44FC 001F   MOVE.W      #$001F,CCR
OUTPUT:
003C10          203C 5555 3333          MOVE.L  #$55553333,D0
003C16          4E40                    TRAP    #0
```

```
003C18          4E41                    TRAP      #1
```

and gives the following output:

```
<RUN>
PC = 00003C06  SR = 00008000  SSP= 00008000   USP=00003FFC
D0 = 0000100D  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
D4 = 0000FC30  D5 = 0000002C  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00000554
A4 = 0000090C  A5 = 00000560  A6 = 00000560
OPCODE WORD NEXT INSTRUCTION = 00007A05

>>
PC = 00003C08  SR = 00008000  SSP= 00008000   USP=00003FFC
D0 = 0000100D  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
D4 = 0000FC30  D5 = 00000005  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00000554
A4 = 0000090C  A5 = 00000560  A6 = 00000560
OPCODE WORD NEXT INSTRUCTION = 000047FA

>>
PC = 00003C0C  SR = 00008000  SSP= 00008000   USP=00003FFC
D0 = 0000100D  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
D4 = 0000FC30  D5 = 00000005  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00003C10
A4 = 0000090C  A5 = 00000560  A6 = 00000560
OPCODE WORD NEXT INSTRUCTION = 000044FC

>>
PC = 00003C10  SR = 0000801F  SSP= 00008000   USP=00003FFC
D0 = 0000100D  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
D4 = 0000FC30  D5 = 00000005  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00003C10
A4 = 0000090C  A5 = 00000560  A6 = 00000560
OPCODE WORD NEXT INSTRUCTION = 0000203C

>>
PC = 00003C16  SR = 00008010  SSP= 00008000   USP=00003FFC
D0 = 55553333  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
D4 = 0000FC30  D5 = 00000005  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00003C10
A4 = 0000090C  A5 = 00000560  A6 = 00000560
OPCODE WORD NEXT INSTRUCTION = 00004E40

>>
PC = 00000984  SR = 00002010  SSP= 00007FFA   USP=00003FFC
D0 = 55553333  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
D4 = 0000FC30  D5 = 00000005  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00003C10
A4 = 0000090C  A5 = 00000560  A6 = 00000560
OPCODE WORD NEXT INSTRUCTION = 00002F07

>>
PC = 00000982  SR = 00002010  SSP= 00007FFA   USP=00003FFC
D0 = 55553333  D1 = 4000544D  D2 = 21FC104D   D3 =00000000
```

```
D4 = 0000FC30  D5 = 00000005  D6 = 00000006   D7 =00000000
A0 = 00004000  A1 = 0000836C  A2 = 00000414   A3 =00003C10
A4 = 0000090C  A5 = 00000560  A6 = 00000560
```
OPCODE WORD NEXT INSTRUCTION = 000060DC

>>

PROGRAM 13.4 ADDRESS ERROR TEST

```
CONSOLEPORT:    EQU     0
LINEFEED:       EQU     $0A
NULL:           EQU     $00
VADDERR:        EQU     $0C
MONITOR:        EQU     $8146

                XREF    INIT, PUTHEX, NEWLINE, PUTSTRING

* INITIALIZE REGISTERS AND CONSOLE PORT
START:          LEA     $8000,SP
                MOVE.L  #ADDRERROR,VADDERR.W
                MOVEQ   #CONSOLEPORT,D7
                JSR     INIT

* TEST PROGRAM
                LEA     $1005, A0
                MOVE.W  2(A0),$7000     * ADDR ERROR -
                                          DATA REF
*               LEA     $1001,A0
*               JMP     (A0)            * ADDR ERROR -
                                          PROGRAM REF

* ADDRESS ERROR SERVICE ROUTINE
PROGCOUNTER:    EQU     10
STATUSREG:      EQU     8
OPCODEWORD:     EQU     6
FAULTADDR:      EQU     2
STATUSWORD:     EQU     0
ASIZE:          EQU     14

ADDRERROR:
                MOVEQ   #CONSOLEPORT,D7
                LEA     ADDERRMSG,A0
                JSR     PUTSTRING
                MOVE.W  OPCODEWORD(SP),D2
                MOVE.L  PROGCOUNTER(SP),A0
* GET VALUE OF PC SAVED ON THE STACK AND SEARCH FOR THE
* OPCODE WORD IN MEMORY
```

```
AGAIN:          CMP.W     -(A0),D2
                BNE.S     AGAIN
                MOVE.L    A0,D0
* PRINT PC AT INSTRUCTION START
                JSR       PUTHEX
                BSR       SPACES

                MOVE.W    STATUSREG(SP),D0      * PRINT SR
                EXT.L     D0
                JSR       PUTHEX
                BSR       SPACES
                MOVE.W    OPCODEWORD(SP),D0
* PRINT OPCODE WORD
                EXT.L     D0
                JSR       PUTHEX
                BSR       SPACES
                MOVE.L    FAULTADDR(SP),D0
* PRINT ADDRESS ACCESSED  WHEN THE FAULT OCCURRED
                JSR       PUTHEX
                BSR       SPACES
                MOVE.W    STATUSWORD(SP),D0
* PRINT STATUS WORD

                ANDI.L    #$1F,D0               * MASK OFF ALL
                                                * UNUSED BITS
                JSR       PUTHEX
                JSR       NEWLINE
                ADDA.W    #ASIZE,SP
                JMP       MONITOR

SPACES:         MOVE.L    A0,-(SP)
                LEA       BLANKS,A0
                JSR       PUTSTRING
                MOVE.L    (SP)+,A0
                RTS

ADDERRMSG:
                DC.B      'ADDRESS ERROR:' ,LINEFEED
                DC.B      'PC          SR          OPCODE WORD ‘
                DC.B      'BAD ADD STATUS WORD',LINEFEED,NULL
BLANKS:         DC.B      '  ',NULL

                END
```

```
<RUN>
ADDRESS ERROR:
PC          SR          OPCODE WORD    BAD ADDR      STATUS WORD
0000091C    00002004    000033E8       00001007      00000015
```

PROGRAM 13.5 SIZING MEMORY

```
XREF INIT,PUTHEX, PUTSTRING, NEWLINE

VBUSERROR:  EQU         $08
NULL:       EQU         $00
MONITOR:    EQU         $8146

* INITIALIZE REGISTERS, EXCEPTION VECTOR, AND CONSOLE PORT

START:
        LEA     $1000,SP        * LET STACK GROW DOWN FROM $1000
        MOVE.L  #ENDMEM,VBUSERROR.W
        MOVEQ   #0,D7           * OUTPUT TO CONSOLE UART
        JSR     INIT            ;initialize UART
        LEA     $1000,A0

* ROUTINE TO TEST MEMORY SIZE AND GENERATE A BUS ERROR

SIZE:   MOVE.W  #7,(A0)+        * WRITE DATA  INTO MEMORY
        BRA.S   SIZE            * LOOP UNTIL BUS ERROR

* BUS ERROR EXCEPTION SERVICE ROUTINE

ENDMEM:
        MOVE.L  A0,D0           * STORE FIRST ADDRESS PAST
                                * RAM MEMORY IN A0
        LEA     MSG,A0
        JSR     PUTSTRING
        SUBQ.L  #8,D0           * DELETE STORAGE FOR RESET VECTOR
        JSR     PUTHEX
        JSR     NEWLINE
        JMP     MONITOR         ;reset the O/S

MSG:    DC.B    'BYTES OF AVAILABLE  RAM: ',NULL

        END
```

<RUN> OUTPUT: BYTES OF AVAILABLE RAM: 00007FF8

A final comment about address and bus errors is necessary. If an address or bus error occurs during exception processing for a bus error, address error, or reset, the processor is halted. Only the external RESET signal can restart a halted processor.

```
PROBLEM 13.8
V1111EMULATOR           EQU                             $2C
VTRACE          EQU     $24
VADDRESS        EQU     $0C

                XREF    PUTHEX,NEWLINE

START:          LEA     $8000,SP
                MOVE.L  #S1111,V1111EMULATOR.W
                MOVE.L  #TRACE,VTRACE.W
                MOVE.L  #ADDRESSERROR,VADDRESS.W

                DC.W    $FFFF
                ORI.W   #$8000,SR
                MOVE.W  D0,D1
                ANDI.W  #$F000,D1
                ANDI.W  #$7FFF,SR

                MOVE.W  #7,$7FFF

PCOUNTER        EQU     14

S1111:          MOVEM.L D0-D1/A0,-(SP)
                MOVEA.L PCOUNTER(SP),A0
                MOVE.W  (A0),D0
                ANDI.W  #$0FFF,D0
                MOVEQ   #0,D1
                JSR     PUTHEX
                JSR     NEWLINE
                ADDI.L  #2,PCOUNTER(SP)
                MOVEM.L (SP)+,D0-D1/A0
                RTE

TRACE           MOVEM.L D0-D1/A0,-(SP)
                MOVEA.L PCOUNTER(SP),A0
                MOVE.W  (A0),D0
                MOVEQ   #0,D1
                JSR     PUTHEX
                JSR     NEWLINE
                MOVEM.L (SP)+,D0-D1/A0
                RTE

PCADDERR        EQU     10
STATUSREGISTER          EQU                             8
OPCODEWORD      EQU     6
FAULTADDRESS            EQU                             2
STATUSWORD      EQU     0
ASIZE           EQU     14

ADDRESSERROR:
                MOVEQ   #0,D1
                MOVE.L  FAULTADDRESS(SP),D0
                JSR     PUTHEX
                JSR     NEWLINE
```

```
                MOVE.W  OPCODEWORD(SP),D0
                EXT.L   D0
                JSR     PUTHEX
                JSR     NEWLINE
                MOVE.W  STATUSREGISTER(SP),D0
                EXT.L   D0
                JSR     PUTHEX
                JSR     NEWLINE
                MOVE.L  PCADDERR(SP),D0
                JSR     PUTHEX
                JSR     NEWLINE

IDLE:           BRA.S   IDLE

                END
```

PROBLEM 13.9

```
V1010EMULATOR           EQU                         $28
VTRAP0          EQU     $80
VTRACE          EQU     $24

                XREF    PUTHEX,NEWLINE

START:          LEA     $8000,SP
                MOVE.L  #EMU,V1010EMULATOR.W
                MOVE.L  #PR,VTRAP0.W
                MOVE.L  #TRACE,VTRACE.W

                DC.W    $A000

                ORI.W   #$8000,SR
                MOVE.W  #1,D0
                MOVE.W  #2,D0
                MOVE.W  #3,D0
                ANDI.W  #$7FFF,SR

ID:             BRA.S   ID

PROGCOUNTER:            EQU                         $14

EMU:            MOVEM.L D0-D1/A0,-(SP)
                MOVEQ   #0,D1
                MOVE.L  PROGCOUNTER(SP),A0
                MOVEQ   #0,D0
                MOVE.W  (A0),D0
                JSR     PUTHEX
                JSR     NEWLINE
                ADDI.L  #2,PROGCOUNTER(SP)
                MOVEM.L (SP)+,D0-D1/A0
                RTE

TRACE:          TRAP    #0
                RTE

PR:             MOVE.L  D1,-(SP)
                MOVEQ   #0,D1
                JSR     PUTHEX
                JSR     NEWLINE
                MOVE.L  (SP)+,D1
                RTE

                END
```

PROBLEM 13.11

Specify what happens when the following code segment runs on a 32K system.

```
BUSERROR    EQU     $08
START:      MOVE.L  #BERR,BUSERROR.W    ;load bus error exception vector
            LEA     $8000,SP            ;start system stack at $8000 (32K)
            MOVE.W  #7,6(SP)            ;put something past 32K - GENERATES BUS ERROR EXCEPTION
BERR:                                   ;bus error service routine
            LEA     26(SP),SP           ;loads address into A7 which is beyond 32K, does not causes EXCEPTION
            RTE                         ;when RTE causes stack access, the value of SP causes another BUS ERROR EXCEPTION - 68000 HALTS
```

RISC/CISC Characteristics

(PowerPC) RISC Technology

References:
Chakravarty and Cannon, Chapter 2
Kacmarcik, Optimizing PowerPC Code

Modern programmers use assembly:
- for handcoding for speed
- for debugging

Common features of CISC:
- many instructions that access memory directly
- large number of addressing modes
- variable length instruction encoding
- support for misaligned accesses

Original goal of RISC (developed in the 1970's) was to create a machine (with a very fast clock cycle) that could process instructions at the rate of one instruction/machine cycle.

Pipelining was needed to achieve this instruction rate.

Typical current RISC chips are HP Precision Architecture, Sun SPARC, DEC Alpha, IBM Power, Motorola/IBM PowerPC

Common RISC characteristics

- Load/store architecture (also called register-register or RR architecture) which fetches operands and results indirectly from main memory through a lot of scalar registers. Other architecture is storage-storage or SS in which source operands and final results are retrieved directly from memory.
- Fixed length instructions which
  (a) are easier to decode than variable length instructions, and
  (b) use fast, inexpensive memory to execute a larger piece of code.
- Hardwired controller instructions (as opposed to microcoded instructions). This is where RISC really shines as hardware implementation of instructions is much faster and uses less silicon real estate than a microstore area.
- Fused or compound instructions which are heavily optimized for the most commonly used functions.
- Pipelined implementations with goal of executing one instruction (or more) per machine cycle.
- Large uniform register set
- minimal number of addressing modes
- no/minimal support for misaligned accesses

NOT NECESSARY for either RISC or CISC

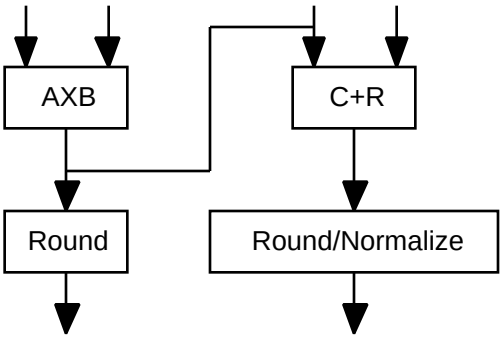
- instruction pipelining
- superscalar instruction dispatch
- hardwired or microcoded instructions

Fused instructions

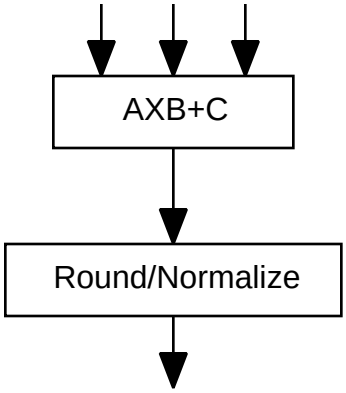
Classical FP multiply
1. Add exponents
2. Multiply significands
3. Normalize
4. Round off answer

Classical FP add
1. Subtract exponents
2. Align decimal points by shifting significand with smaller exponent to right to get same exponent
3. Add significands
4. Normalize
5. Round

Classical instruction



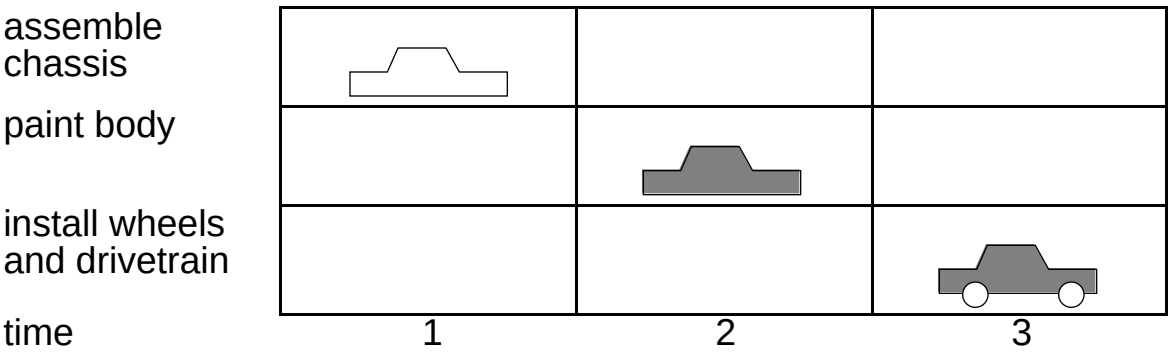
Fused instruction



PIPELINING

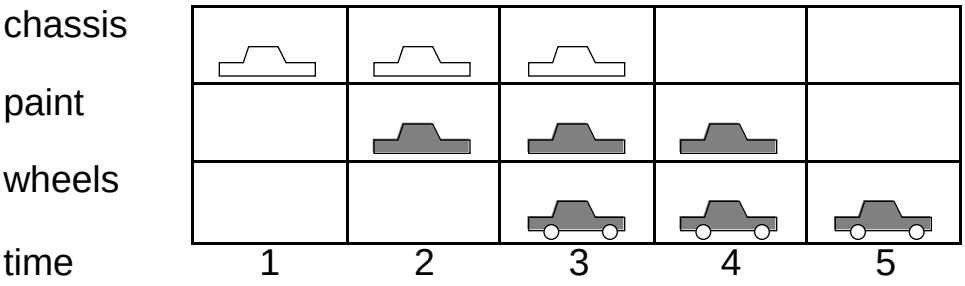
A conventional computer executes one instruction at a time with a Program Counter pointing to the instruction currently being executed. Pipelining is analogous to an oil pipeline where the last product may have gone in before the first result comes out. This provides a way to start a task before the first result appears. The computing throughput is now independent of the total processing time.

Conventional processing



A conventional process would require 9 time units to produce three cars.

Pipelined processing



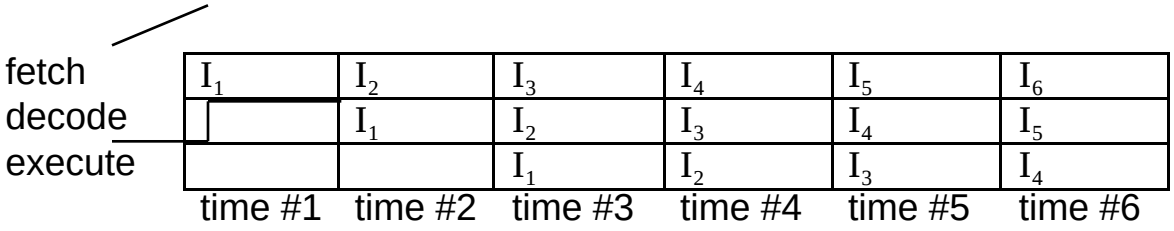
A pipelined process would require 5 time units to produce the same number of cars.

INSTRUCTION PIPELINING

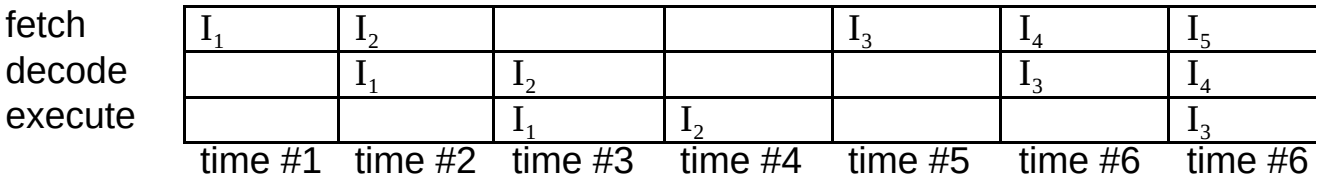
We can apply pipelining to the classical fetch/execute instruction processing. There are three phases to the fetch/execute cycle:

- instruction fetch
- instruction decode
- instruction execute

If we assume these all take one time unit (clock cycle) to execute a three stage pipeline will look like the following.



Pipelining is great in theory but what if there is a branch in your code. You can't determine the next instruction to put into the pipeline until the branch instruction is executed. This can cause a hole, or "bubble" in the pipeline as shown below.



Such bubbles represent performance degradation because the processor is not executing any instructions during this interval.

There are two techniques which can be used to handle this problem with branches:

- delayed branching (as done by an optimizing compiler)
- branch prediction (guess the result of the branch)

normal branch code	delayed branch code
$instruction_0$	$instruction_0$
$instruction_1$	$instruction_1$
$instruction_2$	branch*
branch	$instruction_2$
$instruction_3$	$instruction_3$
•	•
•	•
•	•
$instruction_n$	$instruction_n$

*Delay the instruction originally preceding the branch if it is does not influence the branch. This can be done by an optimizing compiler/assembler. The critical issue is how many independent instructions you have. This is a good technique for pipelines with a depth of 1-2 processes.

Branch prediction, on the other hand, works by “guessing” the target instruction for the branch and marking the instruction as a guess. If the guess was right then the processor just keeps executing; however, if the guess was wrong then the processor must purge the results. The key to this approach is a good guessing algorithm.

The PowerPC uses branch prediction. This approach is very good for FOR and DO/WHILE loops since the branch instruction always branches backwards until the final iteration of the loop. IF/THENs are very bad for guessing and are like flipping a coin with a 50% probability.

Probabilities of branch instructions:

instruction	probability of occurrence	probability of branch
unconditional branch (JMP)	1/3	1
loop closing (FOR and DO/WHILE, Dbcc, etc.)	1/3	~1
forward conditional	1/3	1/2

branch (Bcc, etc.)

The forward conditional branches are the most difficult to guess. The worse case is that we will guess 1/3*1/2 of conditional branches wrong, causing bubbles about 50% of the time..

CISC/RISC tradeoffs

	RISC	CISC
general	very fast, fixed length instruction decode, high execution rate	fewer instructions, size of code is smaller
# of instructions	<100	>200
# of address modes	1-2	5-20
instruction formats	1-2	3+
average cycles/instruction	~1	3-10
memory access	load/store instructions only	most CPU instructions
registers	32+	2-16
control unit	hardwired	microcoded
instruction decode area (% of overall die area)	10%	>50%

RISC cycles
Performance of RISC machine comes from making optimum tradeoff between instruction set functionality (power of each instruction) and clock cycles/instruction.

Program_execution_time = num_instructions_executed * CPI * cycle_time

where num_instructions_executed is dependent upon the pipeline length, CPI is cycles/instruction, and cycle_time is 1/clock_frequency.

PowerPC (PPC)

This is a relatively new architecture with a lot of potential in technical applications.

PowerPC evolution			
IBM POWER architecture	RS.9	1990	
	RS1	1990	
	RSC	1991	
PowerPC	601	1992	<--supports POWER instructions
	603	1993	
	604	1994	
		?	<--first 64 bit PPC's

How does the PowerPC fit the RISC model?

- General purpose registers — 32 general purpose registers (any except GPR0 can be used as an argument to any instruction); 32 floating point registers
- LOAD/STORE architecture — only instructions that access memory are LOAD and STORE instructions
- Limited number of addressing modes

  (I) register indirect;
  (2) register indirect with register index;
  (3) register indirect with immediate index.

  The branch instructions can be

  (I) absolute; (2) PC relative; or (3) SPR (Special Purpose Register) indirect.
- Fixed length instructions — All PPC instructions are 32 bits long.
- No support for misalignments — RISC architecture should not allow misalignments to occur; however, POWER design considerations requiring emulation of other machines allows misalignments.

PPC Data Types

type	size (bits)	alignment
byte*	8	--------
half-word	16	-------0
word*	32	------00
double word†	64	-----000
quad word†	128	----0000
floating point single*	32	------00
floating point double*†	64	-----000

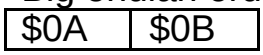
* Most commonly used data types
†64 bit PPC implementation

Alignment

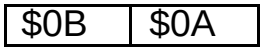
Address must be a multiple of data type size. Bytes are always aligned. Half words must be aligned to even bytes (multiples of 2) just like in the 68000; Words must be aligned to quad bytes (multiples of 4); etc.

Order of bytes

Big endian ordering of 0x0A0B

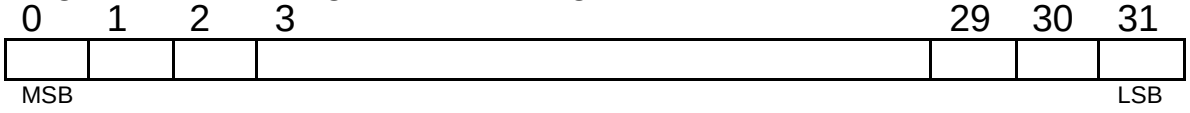


Little endian ordering of 0x0A0B



PPC and 68000 operate in bigendian mode. However, PPC has an option to switch modes.

Big endian ordering of bits in a register:



Super Scalar Implementation

SuperScalar implementation (independent processing units)
PPC 601 has 3 independent execution units so it can actually execute multiple instructions in a single clock cycle. Each execution unit is pipelined. PPC superscalar architecture can execute up to 5 operations/clock cycle.

There are currently two envisioned PPC architectures: 32 and 64 bit. Only the 32 bit implementations have been produced. The PPC architecture does NOT include any i/o definitions.

PPC registers are all 32 bits long (except floating point which are 64 bits long)

PPC consists of three independent processing units

1.	branch processing unit	handles branch instructions
2.	fixed point unit	also called instruction unit
3.	floating point unit	does only floating point instructions

There are three classes of instructions to match the processing units:
1. branch
2. fixed point
3. floating point

All these instructions are 32 bits long and MUST be word aligned.

Because of the Load/Store architecture all computations MUST be done in registers as the operands MUST be loaded into registers BEFORE they can be manipulated/operated on. This typically requires a lot of registers.

PPC Registers

Branch processing unit has three main registers:

Link register — **LR** — contains return address from subroutine calls; contains target address for a branch*

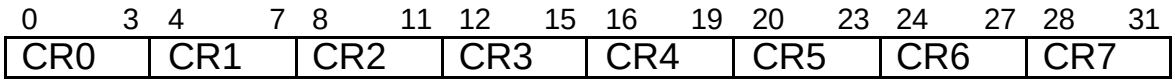
*subroutines can return with a Branch_to_LR instruction

Count register — **CTR** — used for counting loop iterations; treats as Dbcc instructions significantly increasing performance

Counter register — **CTR** — holds number of iterations or a loop; can be used as the final count or as a decrementation counter

Condition register **CR** is the PPC status register

Condition register has 8 4-bit wide condition code fields.



These fields can be specified as a DESTINATION for results of a comparison, or as a SOURCE for conditional branches.

CR0 is usually used for fixed point comparisons

LT	GT	EQ	SO

where SO is the summary overflow. A summary overflow is a “sticky” overflow bit that remains set until reset.

CR1 is usually used for floating point comparisons

FL	FG	FR	FU

FL - floating point less than
FG -floating point greater than
FR - floating point equal
FP - floating point unordered

Fixed point operations with record bit

LT	GT	EQ	SO

LT - negative (<0)
GT -positive (>0)
EQ - zero (=0)
SO - summary overflow

Floating point operations with record bit

FX	FEX	VX	OX

FX - floating point exception summary
FEX -floating point enabled exception summary
VX - floating point invalid operation exception summary
OX - floating point overflow exception

Fixed point processor has most used registers:

32 general purpose registers	**GPR0 - GPR31**	32 bits wide in 32 bit implementations; 64 bits wide in 64 bit implementations; used for all data storage and fixed point operations
Exception register	**XER**	carry, overflow, byte count, and comparison for string instructions

SUPERVISOR MODE REGISTERS:

Machine state register	**MSR**	is processor in 32 or 64 bit mode; are interrupts enabled; bigendian vs. little endian mode
Save/Restore registers	**SSRn**	indicate machine status when an interrupt occurs plus information required to restore state after an interrupt
Processor verification register	**PVR**	READ ONLY. Processor version information.

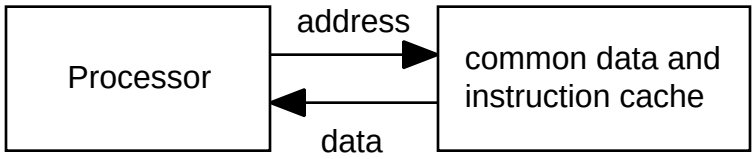
PLUS LOTS MORE!

Floating point processor is similar to fixed point processor:

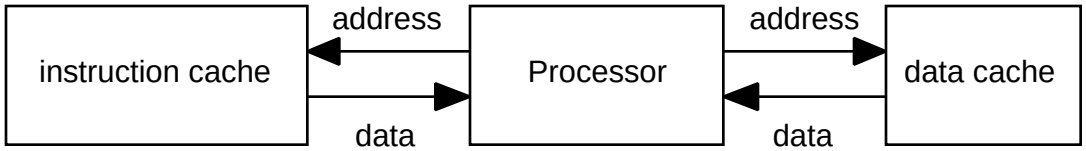
32 floating point registers	**FPR0 - FPR31**	64 bits wide in all implementations; 64 bit registers which are the source and destination for all floating point operations
Floating point status and control register	**FPSCR**	handles floating point exceptions and status of floating point operations; enable bits for fp exceptions; rounding bits to control rounding; status bits to record fp exceptions

PPC Architecture

Many RISC processors use a Harvard architecture; the 601 uses a von Neumann architecture.



von Neumann architecture



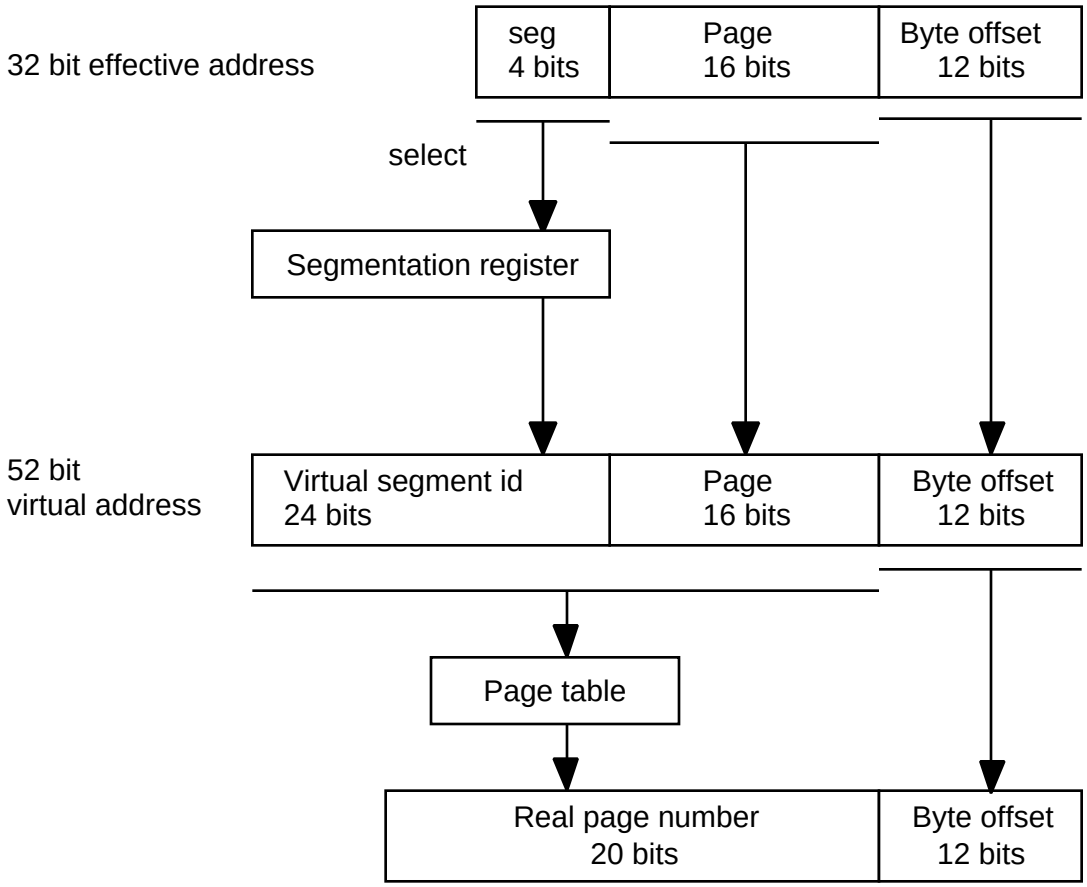
Harvard architecture

Address translation

Effective addresses on the PPC must be translated before they can actually access a physical location. Block address translation takes precedence.

segmented address translation		i/o address
	virtual address	i/o address
		physical address
block address translation	real address	
	i/o address	

Segmented addressing:



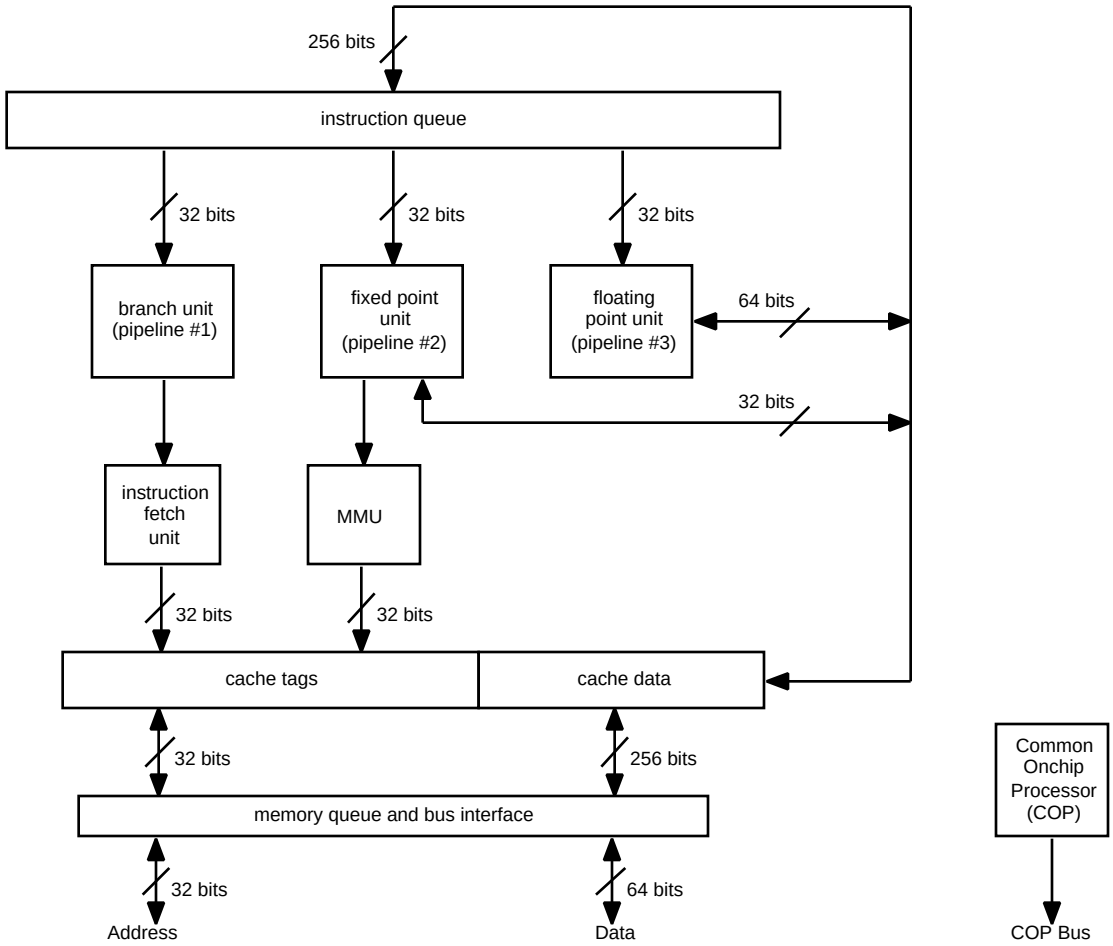
64 bit implementation is VERY different.

Block addressing:

Paged addressing using 4k pages. Block consists of at least 32 pages 128kB up to 65536 pages (256 MB).

The PPC also contains a 64 bit time base register and a 32 bit decrement register which can be used for timing.

Overall PPC 601 architecture:



NOTE: The COP processor controls built-in self test, debug and test features at boot time.

CACHE

Cache is a small memory that acts as a buffer between the processor and main memory. On-chip cache access times are typically 1-2 clock cycles long; access of regular external memory is typically much longer, perhaps 20-30 clock cycles long.

Basic principle of a cache
Locality of reference - Whenever a program refers to a memory address there are likely to be more references to nearby addresses shortly thereafter.

Way cache works
Whenever the main program references a memory location a block of memory containing the referenced address is copied to the cache. The idea is that a lot of following instructions will use information from this cache dramatically speeding up the performance of the processor.

How good a cache works in speeding up computation depends upon:
1. the design of the cache
2. the nature of the executing code

Cache Design and Organization in the PPC

Processor	Size Instruction/ Data	Associativity	Replacement Policy	Line Size (bytes)
601	32K unified	8	LRU	32/64
603	8K/8K	2/2	LRU	32
604	16K/16K	4/4	LRU	32
620	32K/32K	8/8	LRU	64

Notes:

Cache line — the block of memory in the cache that holds the loaded data

Cache tag — pointer from a cache line to the main memory

Line Size — the number of bytes associated with a tag

Associativity — relationship between main memory and the cache. If any block from the main memory can be loaded into any line of the cache, the cache is fully associative. More performance is usually obtained by limiting the number of lines into which a block might reside - this occurs because you have a smaller number of places to look for a particular address. In a two-way associative memory the cache controllerwould only have to examine two tags; in a 4 way four tags; and in an 8-way right tags.

Replacement — When the processor is loading a new block to cache and all the potential lines are full, the cache controller will replace an occupied line with the new data. Common replacement schemes are: first-in first-out (FIFO), least recently used (LRU), and random,

Writeback — (versus store-through) Refers to how the cache contoller handles updates to the information in the cache. In astore-through scheme the stored data is immediately posted to both the cache and main memory. In a store-in (or writeback) scheme only the information in the cache is updated immediately (the line is marked *dirty*) and only updated when the line is replaced in the cache.

Power PC family:

The PPC 601 has three pipelines:

Pipeline #1 (2 stage)	Fetch	Dispatch Decode Execute Predict					
Pipeline #2 (3 stage)	Fetch	Dispatch Decode	Execute	Writeback			
or							
Pipeline #2 (4 stage)	Fetch	Dispatch Decode	Address Generation	Cache (optional)	Writeback		
Pipeline #3 (6 stage)	Fetch	Dispatch	Decode	Execute1 (Multiply)	Execute2 (Add)	Writeback	

*writeback i(or store-in caching) is what happens when the PPC updates data in the cache; posting to main memory is delayed until the line is replaced by the cache unit.

The 603 was designed for portable applications and has four pipelines
1. branch processing unit (2 stage pipeline)
2. fixed point unit (3 stage pipeline)
3. floating point unit (6 stage pipeline)
4. load/store unit (5 stage pipeline)

It also has dynamic power management which controls the processor clock so that only units in use are power up.

The 604 was designed for desktop applications and has two additional integer units giving much improved integer performance. It is in a 304 pin ceramic flat pack with 3.6 million transistors. It dissipates 10 watts at 100 MHz and is based upon 0.5µm CMOS technology.

The embedded versions (4xx, EC403, EC401, etc.) are probably the most economically important.

MAJOR PPC INSTRUCTION GROUPS

- Branch and trap
- Load and store
- Integer
- Rotate and shift
- Floating point
- System integer

Can add suffixes in [ ] to modify instructions

Integer instructions
[o] update FP Exception Register XER
[.] record condition information in CR0

Floating point
[s] single precision data
[.] record condition information in CR1

Branch
[l] (all instructions) record address of following instruction in link register
[a] (some instructions) specified address is absolute

Branch instructions

b[l][a] addr — unconditional branch
b[l][a] BO,BI.addr — conditional branch

[a] indicates that target address is absolute
BO indicates the branch on condition (nine bits and can get complicated)
BI specifies which bit of the CR register is to be used in the test

NOTE: PPC assemblers use b instead of %
Example:
b0000y which indicates decrement the CTR and branch if CTR≠0 and CR[BI]=0.
The y bit encodes hints as to whether branch is likely to be taken.

bcctr [l] BO,BI — branch conditional to count register
Often used to count in loops.

bclr [l] BO,BI — branch conditional to link register
Used for returning from subroutines.

There are probably at least 8-12 extended versions of each basic branch instruction.

lbz rT,d(rA)	load byte and zero; displacement with respect to contents of a source register
load	load from memory location into target register
load with update	add offset afterwards
load indexed	calculate address from two registers
load indexed with update	combination of above
store	write contents of register to memory
store with update	
store with indexed	
store indexed with update	

Example function that performs 64-bit integer addition

```
#    Struct unsigned64
#    {
#       unsigned hi;
#       unsigned lo;
#    }
#
#    unsigned64 add64(unsigned64 *a, unsigned64 *b);
#
# Expects
#    r3 pointer to struct unsigned64 result
#    r4 pointer to unsigned64a
#    r5 pointer to struct unsigned64b
#
# Uses
#    r3 pointer to result
#    r4 pointer to a
#    r5 pointer to b
#    r6 high order word of a (a.hi), high word of sum
#    r7 low order word of a (a.lo), low word of sum
#    r8 high order word of b (b.hi)
#    r9 low order word of b (b.lo)

                lwz     r7,4(r4)        #r7<--a.lo - load word and zero
                                        load the word (words are 32 bits on
                                        the PPC) into r7, uses r4 as its
                                        source
                lwz     r9,4(r5)        #r9<--b.lo - load word and zero
                lwz     r6,0(r4)        #r6<--a.hi load word and zero
                addc    r7,r7,r9        #r7<--sum lo, set CA - add carrying
                lwz     r8,0(r5)        #r8<--b.hi - load word and zero
                stw     r7,4(r3)        #result.lo <--r7 - store word
                adde    r6,r6,r8        #r6<--sum hi with CA - add extended
                stw     r6,0(r3)        #result hi <--r6 - store word
                blr                     #return - branch to link
```