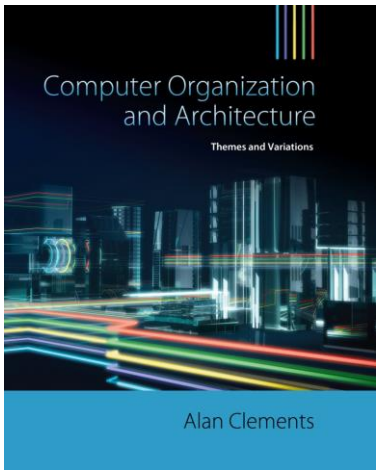
Chapter 4

Computer Organization and Architecture



ISAs Breadth and Depth

- In this presentation we extend our overview of ISAs in both breadth and depth.
- In particular, we look at the role of the stack and architectural support for subroutines and parameter passing.
- We also introduce a class of processors that have both 32-bit and 16-bit ISAs.

HISTORICAL BACKGROUND

Developments in computer architecture have always been influenced by factors such as architectural and technological innovation, the need to maintain backward compatibility with previous members of a family, the changing requirements of users, and fashions in design.

In the 1970s and early 1980s, progress in *commodity* microprocessor architectures was driven by Intel and Motorola.

By the mid-1980s the RISC architectures developed at IBM, Stanford, and Berkeley seemed poised to kill off conventional complex instruction set architectures of the 68K and 80x86 families.

The casual observer could be forgiven for thinking that the conventional CISC such as the Intel IA32 family was nearing the end of its life.

The RISC revolution abandoned complex instruction formats of the 68K and IA32 processors, threw away infrequently used instructions and addressing modes, employed large register sets, and permitted only two memory-based operations, **load** and **store**.

A key feature of RISC machines is the *overlapping* or *pipelining* of instruction execution. As soon as one instruction is read into the computer, the next instruction is fetched from memory while the current instruction is being decoded.

Pipelining thrives on simple, regular instruction formats and doesn't fit well with complex, variable-length CISC instruction formats.

Intel has done a remarkable job in taking its IA32 architecture and applying pipelining techniques to the underlying CISC ISA. Motorola also applied RISC techniques to its 68x00 line.

In the 1980s the arguments in favor of RISC processors appeared to be overwhelming. However, pure RISC machines like MIPS and SPARC didn't sweep all other architectures away because the power of history was too strong.

Too much effort had been invested in ISAs like the Intel IA32 family for people to throw everything away and start again, particularly when an operating system plus one or two software packages costs more than a desktop PC.

Apple abandoned the 68K family in favor of the PowerPC RISC, whereas Intel continued to develop its 80x86 family because of the enormous market provided by the IBM PC and its clones.

Today, the IA32 architecture still dominates the market for PCs and Apple dropped their PowerPC and followed the IA32 path.

THE STACK AND DATA STORAGE

Let's begin by looking at some background issues concerning data storage, procedures, and parameter passing.

High-level language programmers use variables to represent data elements which we can think of a variable as *abstract data cells*. The data cell is abstract because it may hold any type of data element defined by the programmer (e.g., byte, array, record).

As far as the programmer is concerned, the abstract data cell has all the properties of a real memory cell: it can be read from or written to (i.e., data may be assigned to it).

A variable is assigned a name by the programmer. The process of associating the name of a variable with its storage location is called *binding* (binding does much more than simply connecting a name to a variable).

In addition to its name, a variable has a *scope* associated with it.

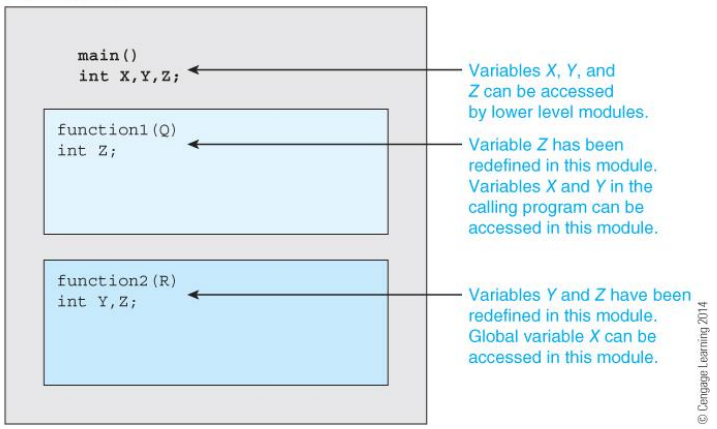
The scope of a variable defines the range of its visibility or *accessibility* within a program.

For example, a variable declared within a procedure might be *visible* within that procedure but *invisible* outside the procedure.

That is, the variable can be accessed inside the procedure, but any attempt to access it outside the procedure would result in an error.

Figure 4.1 illustrates the scope of variables block-structured high-level languages that allows you to define variables that are visible only the current or lower level procedures (or modules). Block structured languages include Algol 60, Pascal, C, Ada and Java.

FIGURE 4.1 The concept of scope



Storage and the Stack

When a language using dynamic data storage invokes a procedure, it is said to *activate* the procedure.

Associated with each procedure and each invocation of a procedure is an *activation record* containing all the information necessary to execute the procedure.

You can regard an activation record as a procedure's view of the world.

Languages that support recursion use dynamic storage because the amount of storage required changes as the program is executed.

Storage must be allocated at *runtime*.

FIGURE 4.2 The activation record

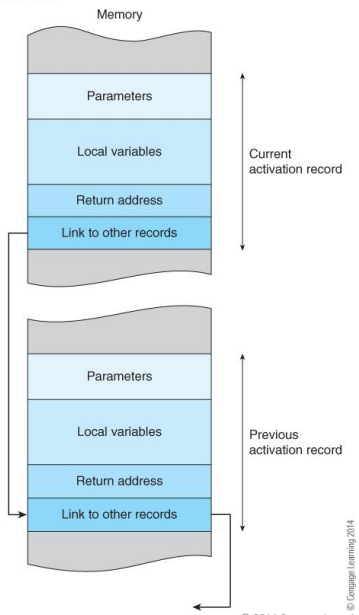


Figure 4.2 illustrates the concept of an activation record.

Temporary storage is needed to evaluate expressions such as

X = (A + B) . (C - D)

because the intermediate result A + B must be stored somewhere while C - D is being calculated.

The activation record described by Figure 4.2 is known as a *frame*.

After an activation record has been used, executing a return from procedure *deallocates* or frees the storage taken up by the record.

We now look at how frames are created and managed at the machine level and demonstrate how two pointer registers are used to implement efficiently activation record creation and deallocation.

Stack pointer and Frame pointer

Two pointers associated with stack frames are the **stack pointer**, SP, and **frame pointer**, FP.

CISCs maintain a hardware SP that is automatically adjusted when a BSR or RTS is executed.

RISC processors like the ARM do not have an explicit SP, although r13 is used as the ARM's programmer-maintained stack pointer by convention.

The stack pointer always points to the top of the stack.

The frame pointer points to the base of the current stack frame.

The stack pointer may change during the execution of the procedure, but the frame pointer will not change. Data in the stack frame may be accessed with respect to either the stack pointer or the stack frame. By convention, r11 is used as a frame pointer in ARM environments and A6 in 68K environments.

The Stack Frame and Local Variables

Procedures often require *local workspace* for their temporary variables.

The term *local* means that the workspace is private to the procedure and is never accessed by the calling program or by other subroutines.

If a procedure is to be made re-entrant or used recursively, its local variables must be bound up not only with the procedure itself, but with the occasion of its use.

Each time the procedure is called, a new workspace must be assigned to it.

If a procedure is allocated a fixed region of workspace, and is interrupted and called by the interrupt routine, any data in fixed locations will be overwritten by the procedure's re-use.

The stack provides a mechanism for implementing the dynamic allocation of workspace.

Two concepts associated with dynamic storage techniques are the *stack-frame* (SF) and the *frame-pointer* (FP).

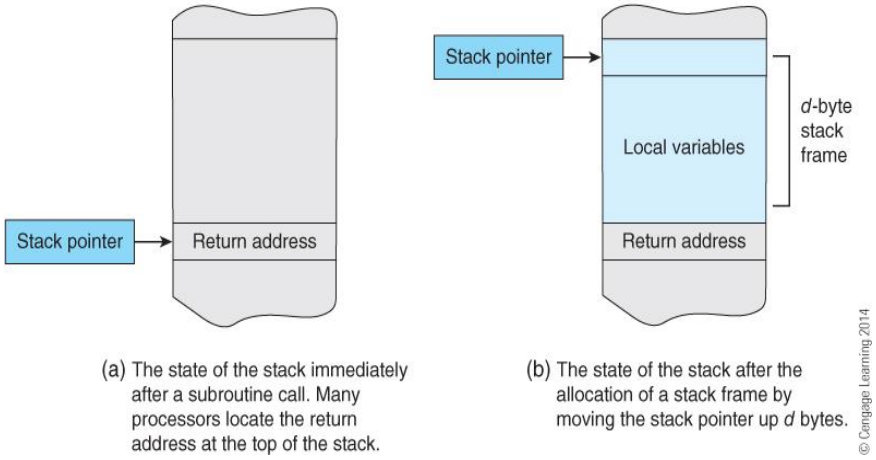
The stack-frame is a region of temporary storage at the top of the current stack.

Figure 4.3 demonstrates how a *d*-byte stack-frame is created by moving the stack pointer up by *d* locations at the start of a subroutine.

We assume that the stack pointer grows up towards low addresses and that the stack pointer is always pointing at the item currently at the top of the stack.

Some stacks point to the next free (empty) element above the stack.

FIGURE 4.3 The stack frame



(a) The state of the stack immediately after a subroutine call. Many processors locate the return address at the top of the stack.

(b) The state of the stack after the allocation of a stack frame by moving the stack pointer up *d* bytes.

Because the stack grows towards the low end of memory, the stack pointer is decremented to create a stack frame;

Reserving 100 bytes of memory is achieved by

SUB r13,r13,#100;move the stack pointer up 100 bytes

Before a return from subroutine is made, the stack-frame is collapsed by restoring the stack pointer with **ADD r13,r13,#100**.

In general, operations on the stack frame are *balanced*; that is, if you put something on the stack frame you have to remove it.

Consider the following simple example of an procedure. Note – this may not be the most efficient code – can you see why?

```
Proc    SUB r13,r13,#16     ; move the stack pointer up 16 bytes
        Code                ; some code
        STR r1,[r13,#8]     ; store something in the frame 8 bytes below TOS
        Code                ; some more code
        ADD r13,r13,#16     ; adios stack frame
        MOV pc,r14          ; time to go home… restore the PC to return
```

The temporary variables on a stack frame can be accessed using the stack pointer.

In Figure 4.4a variable XYZ is 12 bytes below the stack pointer and we access XYZ via address [r13,#12].

Because the stack pointer is free to move as other information is added to the stack, it is better to construct a stack frame with a pointer independent of the stack pointer.

FIGURE 4.4 Accessing variables in the stack frame

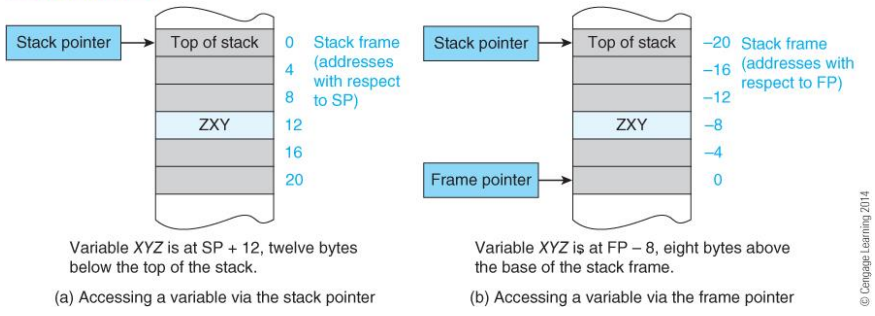
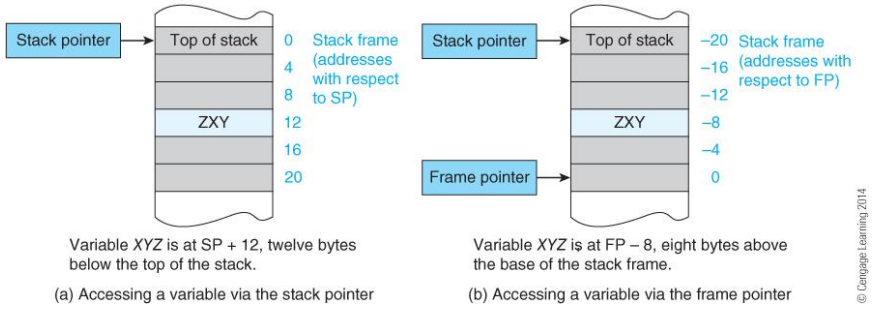


Figure 4.4b illustrates a stack frame with a *frame pointer*, FP, that points to the bottom of the stack frame and is independent of the stack pointer.

The variable can be accessed via the frame pointer at [r11,#-8] if we assume that r11 is the frame pointer.

FIGURE 4.4 Accessing variables in the stack frame



ARM lacks a link instruction that creates a stack frame or an unlink instruction that collapses it.

To create a stack frame you could push the old link pointer on the stack and then move up the stack pointer by *d* bytes by:

```
SUB  sp,sp,#4      ;move the stack pointer up by a 32-bit word
STR  fp,[sp]       ;push the frame pointer on the stack
MOV  fp,sp         ;move the stack pointer to the frame pointer
SUB  sp,sp,#8      ;move stack pointer up 8 bytes (d is equal to 8)
```

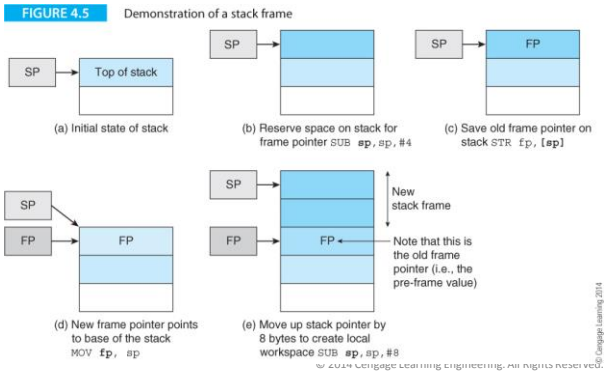
The frame pointer, fp, points at the base of the frame and can be used to access local variables in the frame. By convention, register r11 is used as the frame pointer. At the end of the subroutine, the stack frame is collapsed by:

```
MOV  sp,fp         ;restore the stack pointer
LDR  fp,[sp]       ;restore old frame pointer from the stack
ADD  sp,sp,#4      ;move stack pointer down 4 bytes to restore stack
```


Figure 4.5 demonstrates how the stack frame grows. The old frame pointer appears *twice*; once as the old/previous stack frame on the stack and once as the current stack frame pointing to the base of the stack frame.

We use the pre-decrementing multiple store instruction, **STMFD**, to push both the link register and the frame pointer on the stack with

```
STMFD sp!,{lp,fp}          ;push link register and frame pointer
SUB   sp,sp,#4    ;move stack pointer up 4 bytes
```



Example of an ARM processor Stack Frame

The following demonstrates how you might set up a stack frame. We push a register on the stack, call a subroutine, save the frame pointer and link register, create a one-word frame, access the parameter, and then return to the calling point.

```
        AREA TestProg, CODE, READONLY
        ENTRY  ;This is the calling environment
                        ;subroutine code is on the next slide
                        ;dummy values are used in tracing the code

Main    ADR    sp,Stack        ;set up r13 as the stack pointer
        MOV    r0,#124         ;set up a dummy parameter in r0
        MOV    fp,#123      ;set up dummy frame pointer
        STR    r0,[sp,#-4]!    ;push the parameter
        BL     Sub             ;call the subroutine
        LDR    r1,[sp],#4      ;retrieve the data
Loop    B      Loop            ;wait here (endless loop)
```

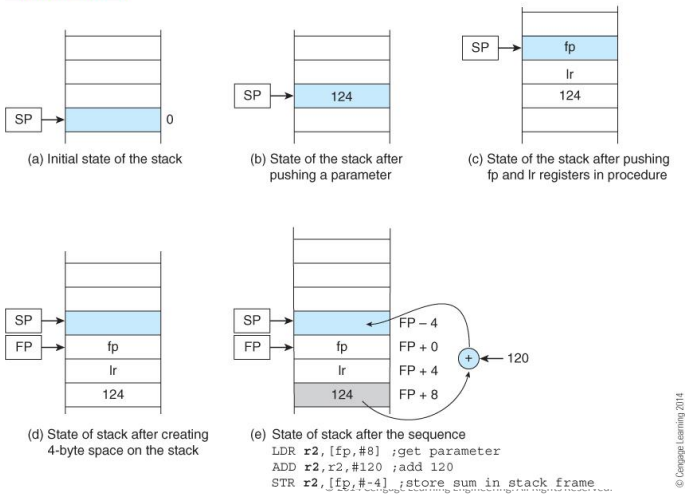
```
Sub     STMFD   sp!,{fp,lr}     ;push frame-pointer and link-register
        MOV     fp,sp           ;frame pointer at the bottom of the frame
        SUB     sp,sp,#4        ;create the stack frame (one word)
        LDR     r2,[fp,#8]      ;get the pushed parameter
        ADD     r2,r2,#120      ;do a dummy operation on the parameter
        STR     r2,[fp,#-4]     ;store it in the stack frame
        ADD     sp,sp,#4        ;clean up the stack frame
        LDMFD   sp!,{fp,pc}     ;restore frame pointer and return

        DCD     0x0000          ;clear memory
        DCD     0x0000
        DCD     0x0000
        DCD     0x0000
Stack   DCD     0x0000          ;start of the stack

        END
```


Figure 4.6 demonstrates the behavior of the stack during the code's execution. Figure 4.6a depicts the stack's initial state. In Figure 4. 7b the parameter has been pushed on the stack. In Figure 4.6c the frame pointer and link register have been stacked by STMFD **sp!**,{fp,lr}.

FIGURE 4.6 The behavior of the stack during the execution of the code



In Figure 4.6d a 4-byte word has been created at the top of the stack. Finally, Figure 4.6e demonstrates how the pushed parameter is accessed and moved to the new stack frame using register indirect addressing with the frame pointer.

FIGURE 4.6 The behavior of the stack during the execution of the code

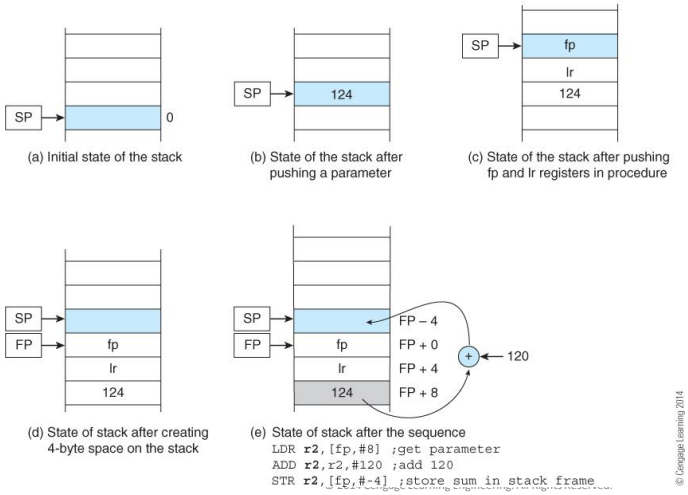


Figure 4.7a provides a snapshot of the output of an ARM processor development system that shows the contents of the registers and the state of the stack after the code has been loaded into the simulator.

In Figure 4.7b we have executed code up to the subroutine call. You can see that the stack pointer (r13) points at 0x08CC and that this location contains 0x7C (the value in r0 pushed on the stack).

In Figure 4.7c we have executed up to the ADD instruction. You can see that the stack pointer is at 0x80C0 and that the link register and old frame pointer have been pushed on the stack.

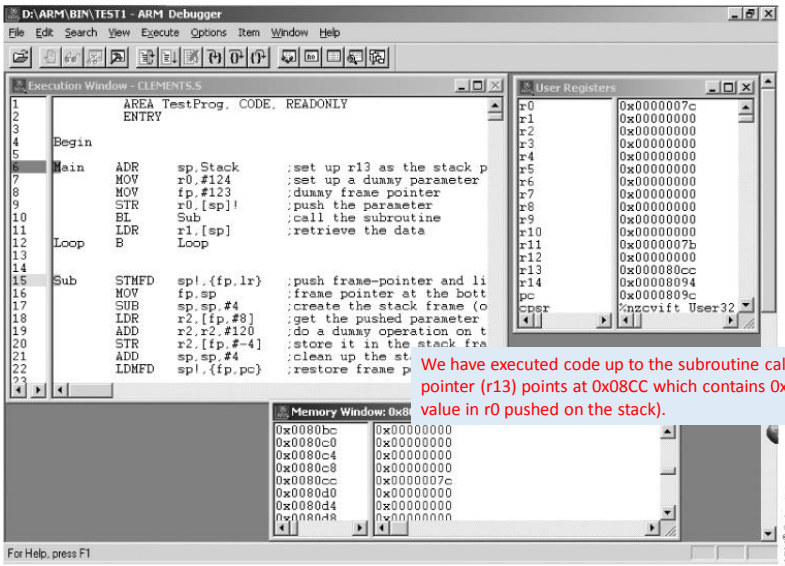
In Figure 4.7d the subroutine has been completed and we have returned to the calling program.

Figure 4.7e shows the state at the completion of the program.

FIGURE 4.7 Snapshots of the state of the registers and memory during the execution of the code

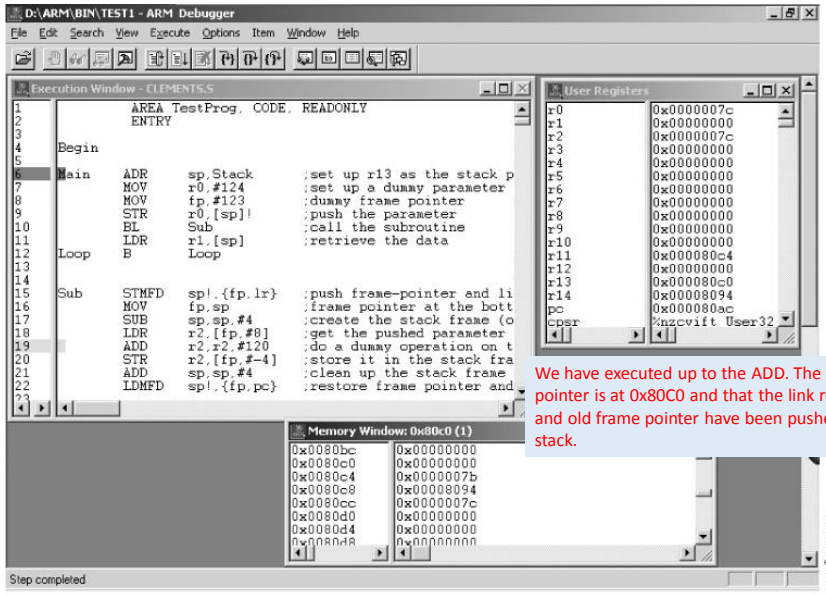
(a) Snapshot of the initial register and memory state on loading the code

FIGURE 4.7 Snapshots of the state of the registers and memory during the execution of the code (*Continued*)



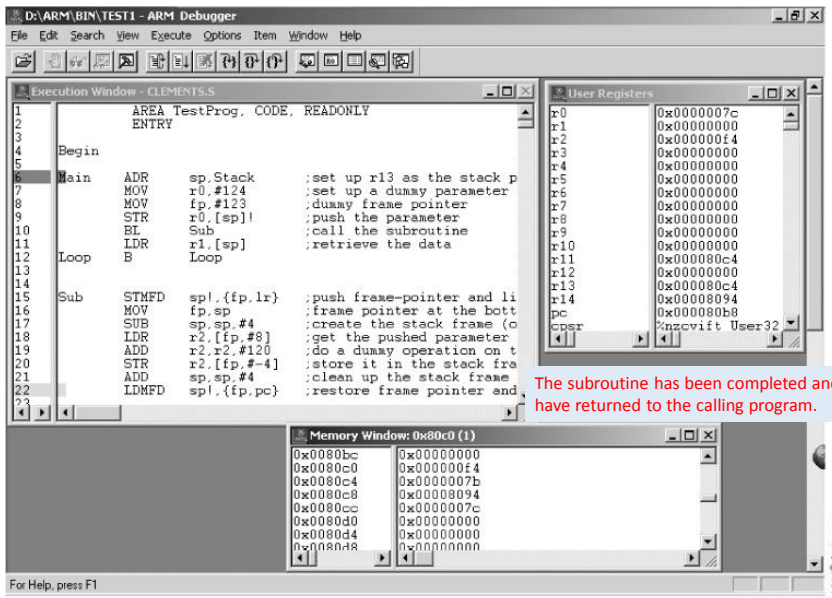
We have executed code up to the subroutine call. The stack pointer (r13) points at 0x08CC which contains 0x7C (the value in r0 pushed on the stack).

(b) Snapshot of the register and memory state on entering the subroutine



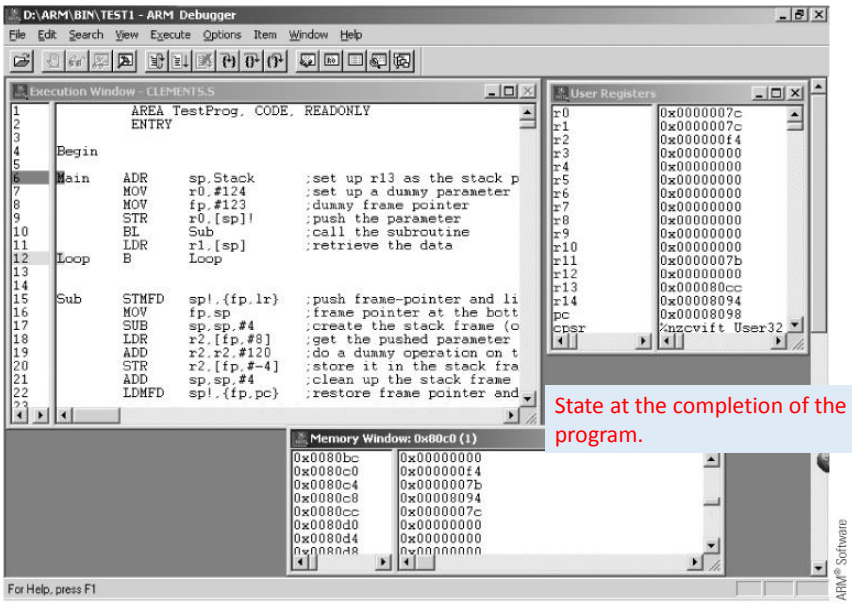
We have executed up to the ADD. The stack pointer is at 0x80C0 and that the link register and old frame pointer have been pushed on the stack.

(c) Snapshot of the register and memory state at the end of the program



The subroutine has been completed and we have returned to the calling program.

(d) Snapshot of the register and memory state before exiting the subroutine



State at the completion of the program.

(e) Snapshot of the register and memory state at the end of the program

Passing Parameters via the Stack

You can pass a parameter to a procedure by *value* or by *reference*. In the former, a copy of the actual parameter is transferred. In the latter, the address of the parameter is passed between the program and the procedure/function.

When passed by value, the procedure receives a *copy* of the parameter. If the parameter is modified by the procedure, the new value does not affect the value of the parameter elsewhere in the program. Passing a parameter by value causes the parameter to be cloned and the clone to be used by the procedure.

When a parameter is passed by reference, the procedure receives a *pointer* to the parameter. There is one copy of the parameter and the procedure accesses this value because it knows the address of the parameter. If the procedure modifies the parameter, it is modified globally.

Passing Parameters via the Stack

Let's examine how parameters are passed to a function when we compile swap(int a, int b) that is *intended* to exchange two values.

```
void swap (int a, int b) /* swaps the value of a and b */
    {  int temp;
    temp = a;            /* copy a to temp, b to a, and temp to b */
    a = b = temp;
    }

void main (void)
    {  int x = 2, y = 3;
    swap (x, y);         /* swap a and b */
    }
```



```
AREA SwapVal, CODE, READONLY
Stop   EQU   0x11                ;code for program termination and exit
       ENTRY

       MOV   sp,#0x1000          ;set up stack pointer
       MOV   fp,#0xFFFFFFFF      ;set up dummy fp for tracing
       B     main                ;jump to the function main

;      void swap (int a, int b)
;      Parameter a is at [fp]+4
;      Parameter b is at [fp]+8
;      Variable temp is at [fp]-4
```



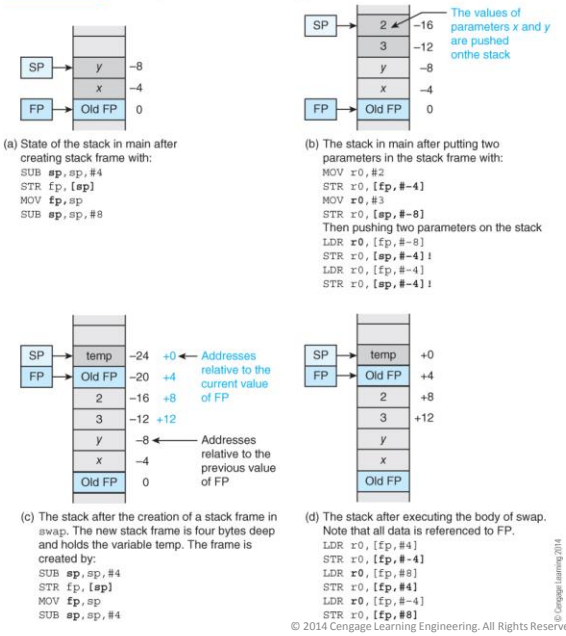
```
swap  SUB   sp,sp,#4          ;Create stack frame: decrement sp
      STR   fp,[sp]           ;push the frame pointer on the stack
      MOV   fp,sp             ;frame pointer points at the base
      SUB   sp,sp,#4          ;move sp up 4 bytes for temp
;     {
;     int temp;
;     temp = a;
      LDR   r0,[fp,#4]        ;get parameter a from the stack
      STR   r0,[fp,#-4]       ;copy a to temp on the stack frame
;     a = b;
      LDR   r0,[fp,#8]        ;get parameter b from the stack
      STR   r0,[fp,#4]        ;copy b to a
;     b = temp;
      LDR   r0,[fp,#-4]       ;get temp from the stack frame
      STR   r0,[fp,#8]        ;copy temp to b
;     }
;                             Collapse stack frame created for swap
      MOV   sp,fp             ;restore the stack pointer
      LDR   fp,[fp]           ;restore old frame pointer from stack
      ADD   sp,sp,#4          ;move stack pointer down 4 bytes
      MOV   pc,lr             ;return by loading link register into PC
```


This code swaps the variables in the stack frame,

When a return is made the stack frame is collapsed and the effect of the swap lost.

The variables in the calling environment are not affected.

FIGURE 4.8 Passing values to a subroutine by value



(a) State of the stack in main after creating stack frame with:

```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#8
```

(b) The stack in main after putting two parameters in the stack frame with:

```
MOV r0,#2
STR r0,[fp,#-4]
MOV r0,#3
STR r0,[sp,#-8]
```

Then pushing two parameters on the stack

```
LDR r0,[fp,#-8]
STR r0,[sp,#-4]!
LDR r0,[fp,#-4]
STR r0,[sp,#-4]!
```

(c) The stack after the creation of a stack frame in swap. The new stack frame is four bytes deep and holds the variable temp. The frame is created by:

```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#4
```

(d) The stack after executing the body of swap. Note that all data is referenced to FP.

```
LDR r0,[fp,#4]
STR r0,[fp,#-4]
LDR r0,[fp,#8]
STR r0,[fp,#-4]
LDR r0,[fp,#8]
STR r0,[fp,#8]
```


In the next example, we pass parameters by reference

```
void swap (int *a, int *b)       /* swap two parameters in calling program */
  { int temp;
    temp = *a;
    *a = *b;
    *b = temp;
  }
void main (void)
  { int x = 2, y = 3;
    swap(&x, &y);   /* call swap and pass addresses of parameters */
  }
.
```

In the next example, we pass parameters by reference. Here's the machine code.

```
        AREA SwapVal, CODE, READONLY
Stop    EQU     0x11                ;code for program termination and exit
        ENTRY
        MOV     sp,#0x1000          ;set up stack pointer
        MOV     fp,#0xFFFFFFFF      ;set up dummy fp for tracing
        B       main                ;jump to main function
;       void swap (int *a, int *b)
;       Parameter a is at [fp]+4
;       Parameter b is at [fp]+8
;       Variable temp is at [fp]-4
.
```



```
swap    SUB     sp,sp,#4    ;create stack frame: decrement sp
        STR     fp,[sp]     ;push the frame pointer on the stack
        MOV     fp,sp       ;the frame pointer points at the base
        SUB     sp,sp,#4    ;move sp up 4 bytes for temp
;       {
```



```
;       int temp;
;       temp = *a;
        LDR     r1,[fp,#4]      ;get address of parameter a
        LDR     r2,[r1]         ;get value of parameter a
        STR     r2,[fp,#-4]     ;store parameter a in temp in stack frame
;       *a = *b;
        LDR     r0,[fp,#8]      ;get address of parameter b
        LDR     r3,[r0]         ;get value of parameter b
        STR     r3,[r1]         ;store parameter b in parameter a
;       b = temp;
        LDR     r3,[fp,#-4]     ;get temp
        STR     r3,[r0]         ;store temp in b
;       }
        MOV     sp,fp           ;Collapse stack frame: restore sp
        LDR     fp,[fp]         ;restore old frame pointer from stack
        ADD     sp,sp,#4        ;move stack pointer down 4 bytes
        MOV     pc,lr           ;return by loading link register contents into PC
```



```
;       void main (void)
;       Variable x is at [fp]-4
;       Variable y is at [fp]-8
main    SUB     sp,sp,#4        ;Create stack frame: move sp up
        STR     fp,[sp]         ;push the frame pointer on the stack
        MOV     fp,sp           ;the frame pointer points at the base
        SUB     sp,sp,#8        ;move sp up 8 bytes for two integers
;       {
;       int x = 2, y = 3;
        MOV     r0,#2           ;x = 2
        STR     r0,[fp,#-4]     ;put x in stack frame
        MOV     r0,#3           ;y = 3
        STR     r0,[fp,#-8]     ;put y in stack frame
```



```
;       swap (&x, &y)           ;call swap, pass parameters by reference
        SUB     r0,fp,#8        ;get address of y in stack frame
        STR     r0,[sp,#-4]!    ;push address of y on stack
        SUB     r0,fp,#4        ;get address of x in stack frame
        STR     r0,[sp,#-4]!    ;push address of x on stack
        BL      swap            ;call swap – save return address in lr
;       }
        MOV     sp,fp           ;collapse frame: restore sp
        LDR     fp,[fp]         ;restore old frame pointer from stack
        ADD     sp,sp,#4        ;move stack pointer down 4 bytes
        SWI     Stop
        END
```


In the function main, the addresses of the parameters are pushed on the stack by means of the following instructions:

```
SUB     r0,fp,#8        ;get address of y in the stack frame
STR     r0,[sp,#-4]!    ;push the address of y on the stack
SUB     r0,fp,#4        ;get address of x in the stack frame
STR     r0,[sp,#-4]!    ;push the address of x on the stack
```

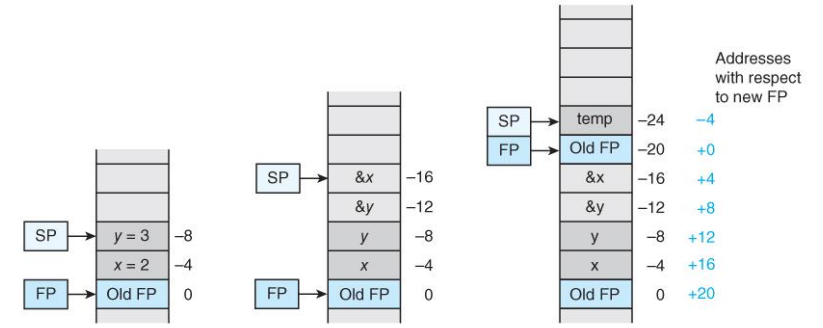
In the function swap, the address of parameter a (i.e., x) is popped off the stack by means of

```
LDR     r1,[fp,#4]      ;get the address of parameter a
```

The operation temp = *a is implemented by

```
LDR     r2,[r1]         ;get the value of parameter a
STR     r2,[fp,#-4]     ;store parameter a in temp in the stack frame
```


FIGURE 4.9 Passing values to a subroutine by reference



(a) State of the stack after
```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#8
MOV r0,#2
STR r0,[fp,#-4]
MOV r0,#3
STR r0,[fp,#-8]
```
in function main

(b) State of the stack after pushing parameter addresses by
```
SUB r0,fp,#8
STR r0,[sp,#-4]!
SUB r0,fp,#4
STR r0,[sp,#-4]!
```

(c) State of the stack after subroutine call and stack frame created by
```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#4
```


Exceptions – an Overview

Exceptions are like subroutines that are *jammed* into code at runtime.

Exceptions use similar call and return mechanisms to subroutines; the major difference being that the call address is supplied by the hardware.

Typically, a processor decodes the exception type and reads a pointer that indicates the start of the exception handling routine. Some processors save the current status word (as well as the return address) because an exception should not alter the processor status.

As well as interrupts, there are page-fault interrupts due to memory access errors, operating system calls, illegal instruction exceptions, and divide-by-zero exceptions. Exceptions are invariably handled by operating system software.

Some processors change their operating mode when an exception occurs. This mode can be a privileged mode in which certain operations are forbidden in order to protect the integrity of the operating system.

Privileged Modes and Exceptions

Exceptions are events that force the computer to stop *normal* processing and to invoke a program called an *exception handler* (usually part of the operating system) to deal with the exception.

At any instant an ARM processor is operating in one of the modes described in Table 4.1.

The five low-order bits of the CPSR define the current mode. The normal operating mode is the *user* mode. A switch between modes takes place whenever an interrupt or exception occurs. Each of these modes has its own saved program status register, SPSR, which is used to hold the current CPSR when the exception occurs.

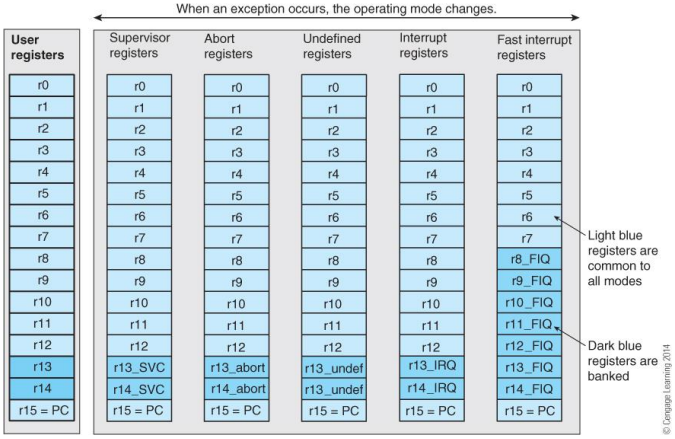
When an exception switches in new registers r13 and r14, *the new register set* (or *bank*) is indicated by the name given in Table 4.1.

TABLE 4.1 The ARM Processor's Operating Modes and the Name of Its Register Sets

Operating Mode	CPSR[4:0]	Use	Register Bank
User	**10000**	**Normal user model**	**user**
FIQ	10001	Fast interrupt processing	_fiq
IRQ	10010	Interrupt processing	_irq
SVC	10011	Software interrupt processing	_svc
Abort	10111	Processing memory faults	_abt
Undef	11011	Undefined instruction processing	_und
System	11111	Operating system	user

Registers in dark blue are banked and associated with specific operating modes. Registers r13 and r14 are replicated in each of the operating modes; for example, if a supervisor exception occurs, the new registers r13 and r14 are called r13_SVC and r14_SVC, respectively.

FIGURE 4.10 The ARM processor's banked register set



When an exception occurs, the ARM processor completes the current instruction (unless the instruction execution itself was the cause of the exception) and then enters an exception-processing mode. The sequence of events that then takes place is:

- The operating mode is changed to the mode corresponding to the exception; for example, an interrupt request would select the IRQ mode.
- The address of the instruction following the point at which the exception occurred is copied into register r14; that is, the exception is treated as a type of subroutine call and the return address is preserved in the link register.
- The current value of the current status processor status register, CPSR, is saved in the SPSR of the new mode; for example if the exception is an interrupt request, CPSR gets saved in SPSR_irq. It is necessary to save the current processor status because an exception must not be allowed to modify the processor status.

- Interrupt requests are disabled by setting bit 7 of the CPSR. If the current exception is a fast interrupt request, further FIQ exceptions are disabled by setting bit 6 of the CPSR.
- Each location in the exception table contains an instruction that is executed first in the exception handling routine. This instruction is normally a branch operation; for example B myHandler. This would load the program counter with the address of the corresponding current exception handler.

Table 4.2 defines the memory locations accessed by the ARM processor's exceptions. Each memory location contains the first instruction of the appropriate exception handlers; this implies, of course, that this table should be in read-only memory.

After the exception has been dealt with by a suitable handler, it is necessary to return to the point at which the exception was called (of course, if the exception was fatal a return is no longer possible).

In order to return from an exception, the information that defines the pre-exception mode must be restored; that is, the program counter and CPSR.

Unfortunately, returning from an exception is not as trivial a matter as it might seem. If you restore the PC first, you are still in the exception-handling mode. On the other hand, if you restore the processor status first, you are no longer within the exception-handling routine and there is no way in which you can restore the CPSR.

TABLE 4.2 Exception Vectors

Exception	Mode	Vector Address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

You can't use a *normal* sequence of operations to return from an exception because it involves a change of operating mode.

Two exception return mechanisms are provided, one for the case in which the return address has been stored in the banked r14, and the other for the case in which the return address has been pushed on the stack. Moreover, the return mechanism depends on the type of exception being handled.

If you are returning from an exception where the return address is in the link register, you can execute instructions described in Table 4.3, where MOVS and SUBS are special versions of the normal instructions used when the destination register is the pc.

You have to modify the value of the pc when returning from an IRQ, FIQ, or a data abort. In the former case, the pc has to be wound back by 4. In the latter case the pc has to be wound back by 8 in order to repeat the instruction that was faulted.

Exception type	Instruction to return to user mode
SWI, undefined instruction	MOVS pc,r14
IRQ, FIQ	SUBS pc,r14,#4
Data abort to repeat the faulted instruction	SUBS pc,r14,#8

Table 4.3 ARM return from exception operations

If the exception handler has copied the return address on the stack, you have to use a slightly different mechanism. Under normal circumstances, you would return from a subroutine with a stacked pc by means of an instruction such as

LDMFD **r13!**, {r0-r4, pc}

where r0-r4 is the list of registers to be restored. If you wish to unstack the saved registers and restore the CPSR at the same time, you have to use the special version of this instruction

LDMFD **r13!**, {r0-r4, pc}^ ;restore r0 to r4, return and **restore CPSR**

The "^" symbol after the register list indicates that the CPSR is to be restored at the same time the program counter is restored. The program counter was not modified at the point at which it was restored. You have to modify the PC *before* you stack it!

MIPS: ANOTHER RISC

Having looked at ARM, we take a brief look at another processor – MIPS

MIPS is a RISC architecture developed by John Hennessy at Stanford University in 1980 to exploit the best aspects of RISC philosophy in an efficient 32-bit processor.

MIPS has gone through several generations and is available in 64 bit versions. MIPS is important because it has been widely used to support the teaching of computer architecture.

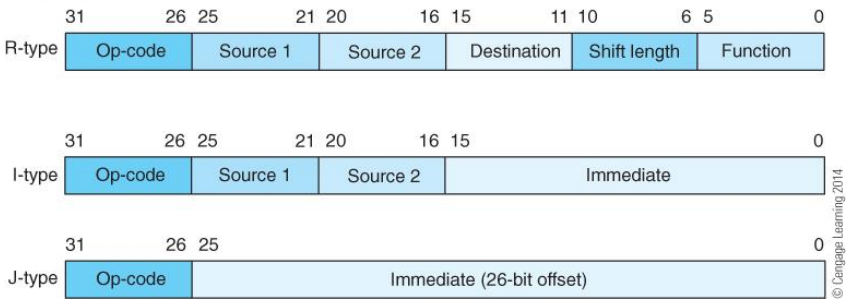
MIPS makes an interesting contrast with the ARM processor. MIPS is found in a wide range of embedded and mobile applications and in some games system; for example, PlayStation.

MIPS has a classic 32-bit load and store ISA and 32 general-purpose registers. Register r0 is unusual because it holds a zero and cannot be changed. This is an important feature of MIPS because it allows the programmer easy access to zero and an ability to *suppress* a register in an instruction (use r0 and you get zero).

Figure 4.11 illustrates three MIPS instruction formats: R-type that specifies register-to-register operations, I-type that provides a 16-bit literal operand, and J-type used for direct jump instructions.

There is also a C-type for coprocessor operations that we do not discuss here.

FIGURE 4.11 MIPS instruction formats



The R-type instruction provides a register-to-register data processing operation. The most significant difference between MIPS and ARM processors is that MIPS can specify one of 32 registers, whereas ARM provides only 16 registers.

A typical R-type instruction is add **r1**,r2,r3. MIPS lacks two important ARM processor mechanisms, conditional execution and the ability to shift the second operand.

The I-type instruction concatenates three fields from the R-type instruction to create a 16-bit literal field to provide a constant in operations like *add immediate* or an offset in register indirect addressing modes. The 16-bit literal which may be signed or unsigned permitting a range of -32,768 to +32,767 or 0 to 65,535. The literal cannot be scaled.

A typical I-type operation is addi **r1**,r2,4. MIPS appends an i to the opcode to indicate literal, whereas the ARM processor uses the # symbol to prefix literal. These differences refer to the assembler grammar and not the ISA of the processors.

Because MIPS uses 16-bit literals, depositing a 32-bit word into a register is easily done by loading two consecutive literals.

A *load upper immediate* instruction, lui, deposits a 16-bit literal into the upper-order 16-bits of a register and clears the lower-order 16 bits to zero; for example lui **$1**,0x1234 loads register r1 with 0x12340000.

A logical OR with a 16-bit immediate operand can now be used to access the lower-order 16 bits; for example, ori **$1**,0xABCD will set r1 to 0x1234ABCD.

The J-type instruction format is unconditional jumps and provides a 26-bit literal that is used to construct a branch offset.

Because MIPS is word (32-bit) oriented, the branch offset is shifted left twice before using it to provide a 28-bit byte range of 256 Mbytes.

The MIPS register set is conventional and, apart from r0 that is fixed at 0, has no special-function registers .

MIPS assembly language uses $0,$1, … rather than r0,r1, … as the name of registers.

Table 4.4 describes the MIPS register set and gives the alternate registers names used by programmers.

MIPS load and store instructions are lw (load word) and sw (store word). Addressing modes are minimal and MIPS provides only a register indirect with offset addressing mode; for example lw **$1**,16($2) implements [$1] ← [16+[$2]].

MIPS lacks the complex addressing modes of CISCs and the ARM processor's block move instructions. However, direct memory addressing is possible if you use register r0 (because that forces a 16-bit absolute address), and program counter-relative addressing is supported.

Conditional Branches

MIPS handles conditional branches in a markedly different way from the ARM processor.

Recall that an ARM processor branch depends on the state of processor condition code bits set or cleared by a previous instruction.

MIPS provides *explicit* compare and branch instructions; for example, beq r1,r2,label compares the contents of register r1 with r2 and branches to label on equality.

MIPS lacks the set of 16 conditional branches provides by CISC processors (and the ARM processor) and implements only

```
beq  $1,$2   ;Branch on equal
bne  $1,$2   ;Branch on not equal
blez $1,$2   ;Branch on less than or equal to zero
bgtz $1,$2   ;Branch on greater than zero
```


An interesting MIPS instruction is the *set on condition*; for example, the *set on less than* instruction slt **$1**,$2,$3 performs the test [$2] < [$1] and then sets $1 to 1 if the test is true and to 0 if the test is false.

This turns a Boolean condition into a value in a register that can later be used by a conditional branch or as a data value in an operation.

A typical example of the use of slt is

```
slt  $1,$2,$3          ;if $2 < $3 THEN $1 = 1 ELSE $1 = 0
bne  $1,$0,Targe       ;branch on $1 not zero (that is, branch on $2 < $3)
```

There is also an sltu operation that performs the same operation on unsigned numbers, and slti and sltui versions that have immediate operands.

MIPS Data Processing Instructions

MIPS data processing operations are generally very similar to the ARM processor's data processing instructions.

One small difference is that MIPS provides explicit shift operations that provide either a fixed length shift with a literal shift field, or a dynamic shift with a register shift field; for example,

```
sll  $1,$2,4       ;Shift $2 left 4 places and put the result in $1
sllv $1,$2,$3      ;Shift $2 left the number of places in $3, result in $1
```

Note that a different instruction is required for static and dynamic shifts. This is a feature of the assembler rather than the ISA.

DATA PROCESSING AND DATA MOVEMENT

Now we look at some of the aspects of data movement ranging from packing and shifting data elements, to processing groups of bits, to checking that data elements are within the correct bounds. We also look at processors other than ARM.

The point of this section is to demonstrate the variety in the approach to computer ISA design.

The most frequent computer operation is *data movement*. Computers have load and store instructions and register-to-register data transfers.

Sometimes you have to do more than copy data from one place to another – modify the order of the bytes in a 32-bit word as they are moved or move data from consecutive memory locations to consecutive *odd* or *even* locations.

The figure below illustrates some variations in the move instruction beginning with the basic transfer of data between a register and memory, figures (a) and (b).

All processors permit register-to-memory, memory-to-register, and register-to-register moves. Few microprocessors permit direct memory-to-memory moves.

(a) Move memory to register

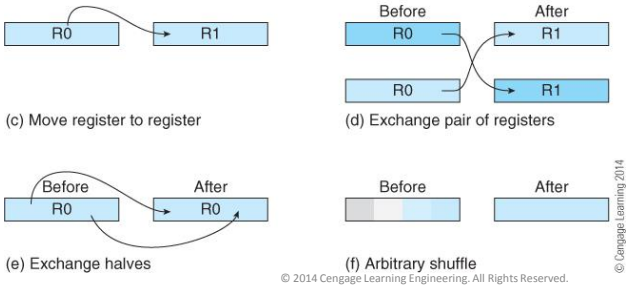
(b) Move register to memory

Some processors can exchange the contents of two registers; for example, EXG X,S swaps the X and S registers.

Some can swap one field of a register with; for example, SWAP X exchanges the two halves of a register.

Figure (d) illustrates the exchange instruction and figure (e) describes an instruction that swaps over the two halves of a register.

You could devise an instruction that allows you to arbitrarily shuffle the bytes of a register as (f) demonstrates.

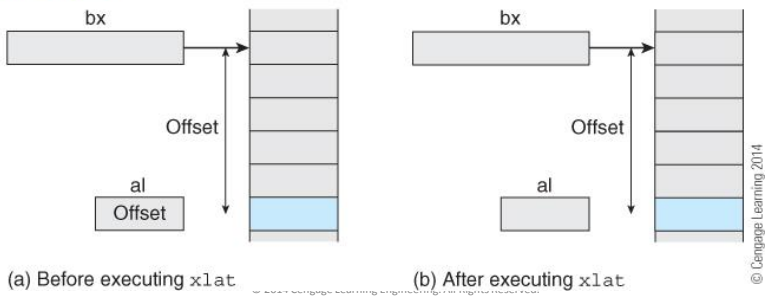


Example of a Special Data Movement Operation

Figure 4.12 describes an IA32 instruction xlat (translation) with no parameters. It employs the 8-bit al register and 16-bit base register bx.

Base register bx points to memory and al contains an 8-bit offset. When xlat is executed, the contents of al are added to bx to give an effective address. The 8-bit operand at this address is loaded into al. The offset is used to look up the data element in a table and then data replaces the offset.

FIGURE 4.12 Effect of the `xlat` instruction



xlat demonstrates the strengths and weaknesses of the CISC philosophy. A single instruction performs an operation normally requiring two operations (i.e., add the index to the base register and perform a register-indirect move). xlat is instruction because it doesn't require operands (the use of the bx and al registers is implicit).

xlat demonstrates the weakness of the CISC philosophy. It is used in one specific application and is inflexible (the operand size is fixed and it can be used only with the al and bx registers).

Indivisible Exchange Instructions

Some data move instructions provided by both CISC and RISC processors look, at first glance, rather strange; for example, IA32 processors provide a *compare and exchange* instruction cmpxchg that uses three operands (one implicit and two explicit).

Its format is cmpxchg reg,reg or cmpxchg mem,reg. This instruction compares the al, ax, or eax accumulator with the first operand and sets the zero flag if they are equal, and then copies the second operand into the first.

If the accumulator and first operand are not equal, cmpxchg copies the first operand into the accumulator. We can describe the effect of cmpxchg **bx**,cx as

IF [ax] = [bx] THEN [z] ← 1, [bx] = [cx]
ELSE [z] ← 0, [ax] = [bx]

Indivisible Exchange Instructions

This instruction is indivisible because it includes two operations, a test followed by one of two actions, and the instruction cannot be interrupted.

Such an instruction is needed by distributed systems to ensure that an external device can ask whether a resource is free and then claim it, without another device making the same request between the asking and receiving phase of the instruction.

Instruction sets often include synchronizing operations as well as data processing operations.

Double-precision Shifting

Shift operations move all the bits of a register one or more places left or right; consequently, the maximum number of bits you can perform is equal to the length of a register.

Sometimes you have to perform a shift over a larger number of bits; for example, when performing extended-precision arithmetic.

Some processors do provide an extended shift in which the carry bit is included in this shift, allowing you to implement a multiple-precision shift in which the bit shifted out of one register is shifted into the carry, and then into the second register taking part in the shift.

Example (with 4-bit registers)

0010 1101

Shift one bit left

0101 1010 (the bit shifted out of one register is shifted into the other)

Double-precision Shifting

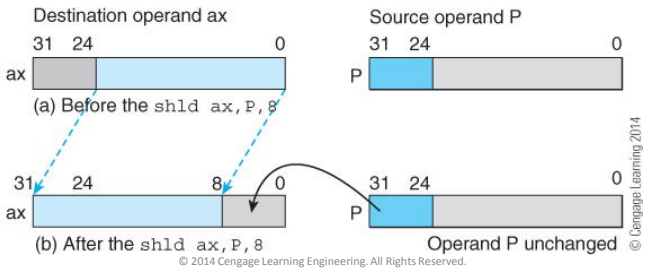
The IA32 provides two double-precision instructions shld and shrd (shift left double and shift right double) that take a pair of operands and shift both simultaneously.

The left-shift forms of the instruction are

shld **operand1**,operand2,immediate ;immediate defines number of shifts

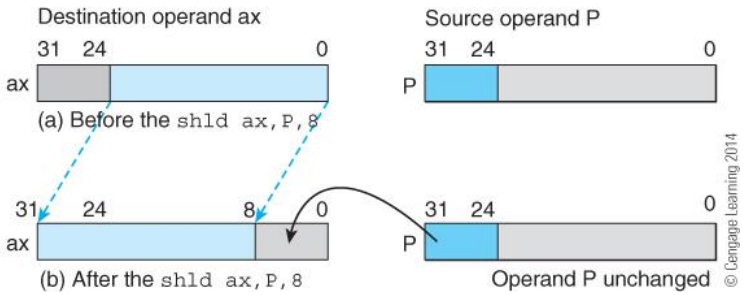
shld **operand1**,operand2,cl ;register cl allows dynamic shifts

FIGURE 4.13 Using the `shld` instruction



In Figure 4.13 the bits of P are shifted left eight places and become the low-order 8 bits of ax. Bits 0 to 23 of ax are also shifted left 8 places.

FIGURE 4.13 Using the `shld` instruction



This double length shift instruction can be used to pack data from several sources into a single register.

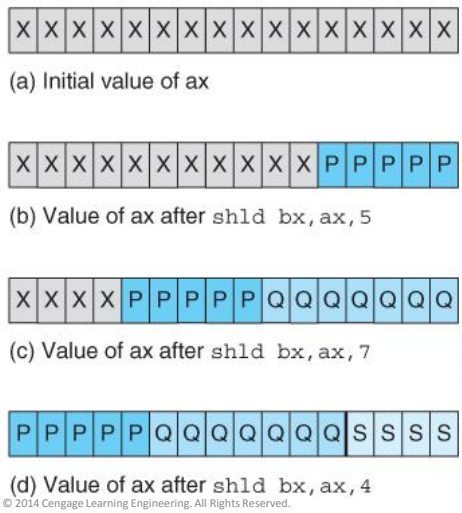
Suppose we wish to pack register bx with 5 bits from memory location P, 7 bits from location Q, and 4 bits from location R.

These bits are packed in the order PQR, where P is the most-significant 5 bits. We can use:

```
mov  ax,P      ;read the high-order bits from P into the accumulator
shld bx,ax,5   ;copy the high-order 5 bits from ax into bx
mov  ax,Q      ;read the middle bits from Q into the accumulator
shld bx,ax,7   ;copy the middle-order 7 bits from ax into bx
mov  ax,R      ;read the low-order bits from R into the accumulator
shld bx,ax,4   ;copy the low-order 4 bits from ax into bx
```


Figure 4.14 illustrates the effect of these instructions.

FIGURE 4.14 Using `shld` to pack data



Pack and Unpack Instructions

Packing and unpacking data implies moving multiple data elements into a single register or memory location (packing), or moving one data element into multiple registers or memory locations.

Let's look at an example from the 68K ISA that implements PACK and UNPK instructions.

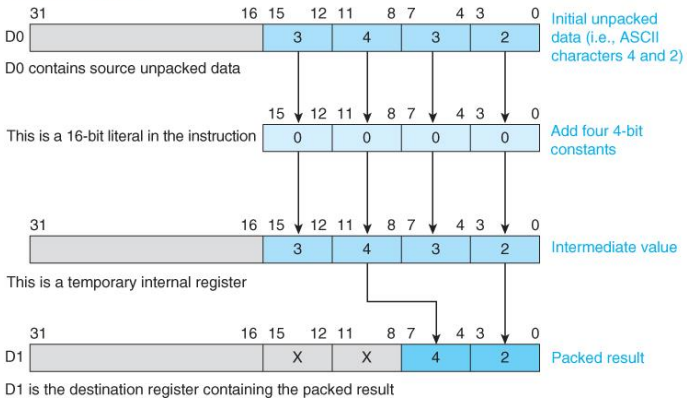
Both these instructions act on the lower-order 16-bits of a 32-bit register.

Figure 4.15 illustrates the action of PACK D0,**D1**,#literal.

The PACK instruction takes the four 4-bit values in register D0 (in this case 3432_{16}) and converts them to the two 4-bit values (in this case, 42_{16}).

This instruction to facilitates the conversion between unpacked ASCII characters and packed BCD data.

FIGURE 4.15 The PACK instruction



In this example the two ASCII characters 4 and 2, corresponding to codes 34_{16} and 32_{16}, respectively, are converted into the BCD equivalent 42_{10}. The conversion process allows a 4-bit literal to be added to each of the source 4-bit words. In this case, the constants are all zero.

FIGURE 4.15 The PACK instruction

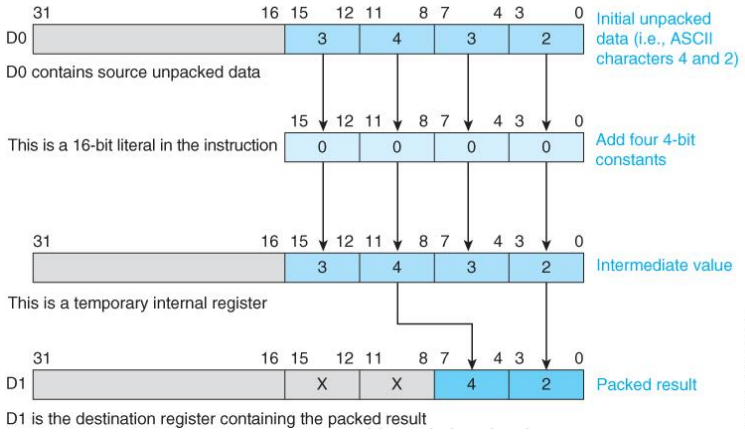
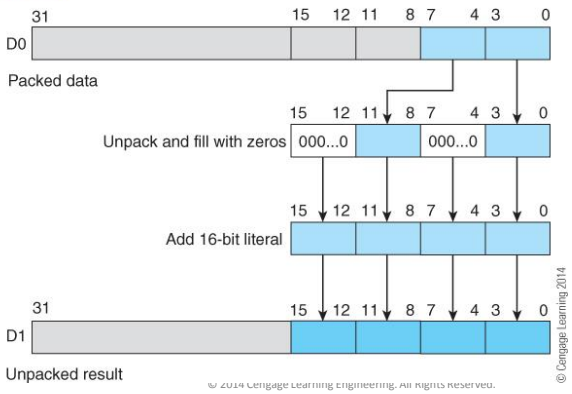


Figure 4.16 describes the inverse of the PACK instruction, UNPK, that takes two hexadecimal nibbles in the low-order byte of a word and converts them into two 8-bit values.

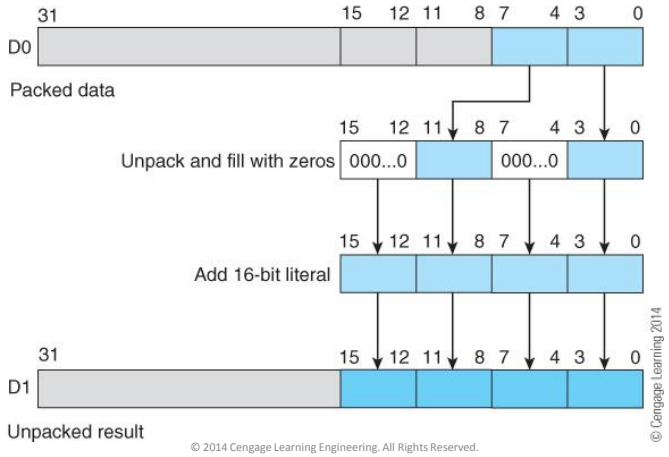
In this case the two nibbles are moved into consecutive bytes and a constant added to the result.

FIGURE 4.16 The UNPK instruction



If you are converting BCD values to ASCII character codes, you execute the instruction UNPK D0,**D1**,#$3030 because a BCD digit is converted to its corresponding ASCII code by adding 30_{16}.

FIGURE 4.16 The UNPK instruction



BOUNDS TESTING

When working with data structures such as arrays and tables, you need to know whether the element you are accessing falls within the array.

An array access error occurs when the index (location) of an element is incorrectly computed at run time.

A problem can arise if the value of the array element is computed incorrectly and a data value is accessed outside the range of the array.

Some high-level languages test that the subscript of an array being accessed is within its correct bounds (C does not provide such testing).

The 68020 implements a *bounds checkin*g operation, CHK2 that determines whether an array subscript is within its correct range.

If an out-of-range condition is detected, the operating system is invoked to deal with the situation; that is, a trap or exception is called.

Typically, an array subscript is compared against its upper and lower limits using two tests and two conditional branches to determine whether the address is within range. We can do the same with the single 68020 instruction CHK2 as follows.

```
        LEA   Array,A0          *Register A0 contains the base address
                                *of the array
        ADDA  D0,A0             *Element index in D0 is added to the base
*                               *and A0 now points to
*                               *the required element
        CHK2.L Bounds,A0        *Do a bounds check on pointer A0
        MOVE  (A0),D1           *Read the required element

Bounds  DC.L  Lower             *Store the lower bound in memory
        DC.L  Upper             *followed by the upper bound
```


In this case we require only one instruction to perform both the *upper* and *lower* bounds check. **CHK2.L Bounds,A0** compares the value in A0 first with the lower bound at the address given by Bounds and then compares the value in A0 with the address given by Bounds+4.

Here the bounds are 32-bit, four-byte values. If the value in A0 is within range, nothing happens.

If it is outside the range defined by the bounds, an exception is generated and the operating system must deal with the recovery. The 68020 also provides a

CMP2 instruction that has the same format as the CHK2 instruction but which sets the carry flag to signal an out-of-range error.

Figure 4.17 illustrates the relationship between the bounds specified in the CHK2 instruction and the range of valid values. We test register A6 against a pair of bounds. In the first two examples, the range is unsigned.

FIGURE 4.17 Examples of the use of the CHK2 instruction

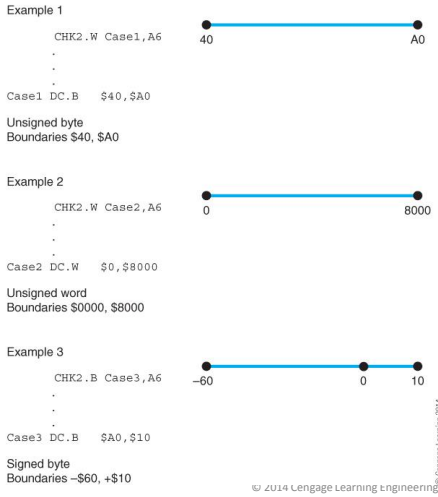
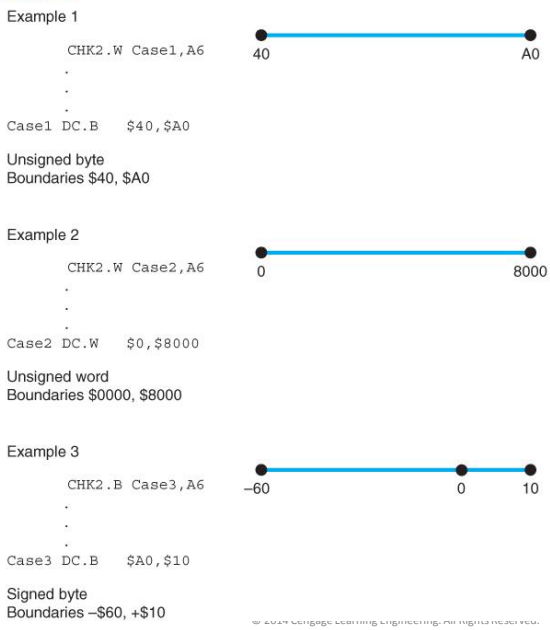


FIGURE 4.17 Examples of the use of the CHK2 instruction



BIT FIELD DATA

The *bit field* is a data structure that is an arbitrary string of bits of any length.

You can use bit fields to represent information that doesn't fit into a 8-, 16-, 32-, or 64-bit package like characters, integers and floating-point values; for example, a 19-bit bit field might represent a packed data value consisting of three fields of three bits, seven bits, and nine bits.

Equally, it may represent a line of pixels in an image.

BIT FIELD DATA

There's no fundamental reason why we cannot consider memory as a long string of bits.

Bit fields are not widely implemented because of the additional complexity they impose on the underlying hardware.

Since memory is physically byte-oriented with 8-, 16-, 32-, or 64-bit buses, an access to a bit field that spans several words may require multiple consecutive memory accesses which degrades performance.

Because a bit field is nothing more than a string of consecutive bits, we can define a bit field in terms of two parameters – its *width* or length *w*, and its location in memory *q*.

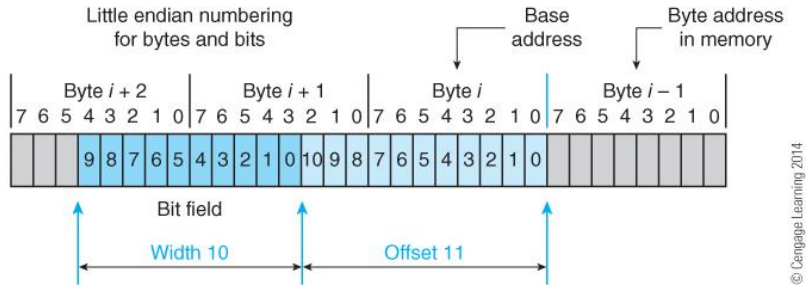
The value of *q* is, of course expressed in bits; for example, we could define a 56-bit bit field *x* as beginning 92,345 bits away from the first bit in memory and extending from bit 92,346 to bit 92,401.

An alternative way of specifying bit fields involves a compromise between bits and bytes – it uses a byte address to specify a location in memory and then a bit offset from this location to specify the bit field's position with respect to the designated byte.

Figure 4.18 illustrates a bit field specified by a byte address plus an offset; the bit field starts 11 bits from bit 0 in byte *i* in memory and is 10 bits wide.

We have numbered the bytes in memory from right-to-left and used a little endian arrangement for both bytes and bits.

FIGURE 4.18 The bit field



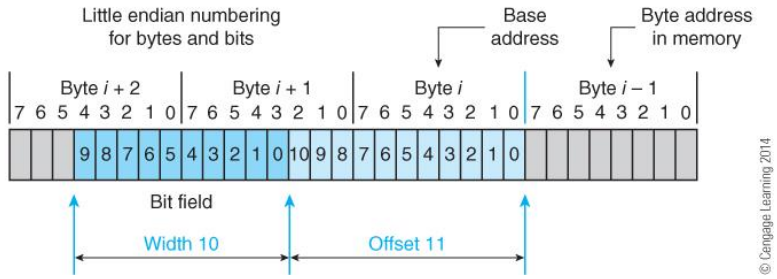
The structure in Figure 4.18 is *little endian consistent*.

The bytes are numbered from the least-significant byte (on the right) as are the bits of a byte.

The offset of the bit field from byte *i* begins at bit 0 of that byte.

The offset bits are also numbered right-to-left, as are the bits of the bit field.

FIGURE 4.18 The bit field



68020 bit fields

The 68020 microprocessor was the first CISC processor to support 32-bit fields.

FIGURE 4.19 The 68020's bit field organization

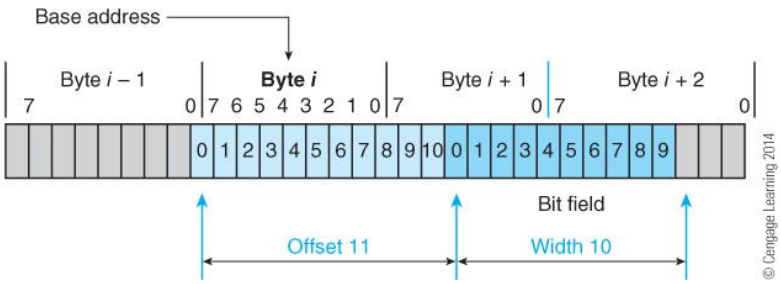
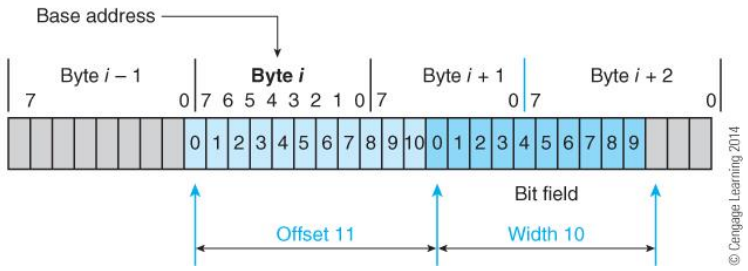


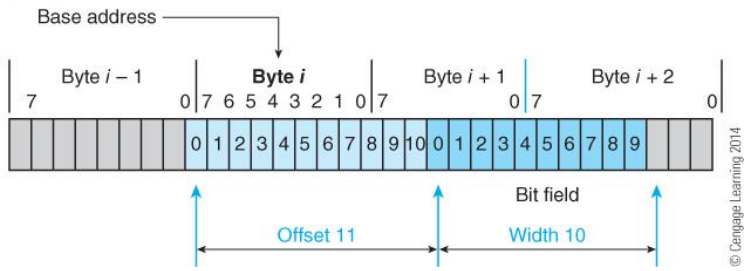
Figure 4.19 demonstrates that the bit field location is defined with respect to the most-significant bit of byte *i*, (byte *i* is called the *base byte* and is the effective address of the bit field specified in instructions) and the bits of the bit field are numbered in the reverse sense to the bits of a byte; that is, the bit field follows the big endian numbering convention.

FIGURE 4.19 The 68020's bit field organization



The least-significant bit of a bit field begins at bit 7 of the base byte and, let's repeat this, the bits of a bit field are numbered in reverse order with respect to the bits of a byte.

FIGURE 4.19 The 68020's bit field organization



The 68020 allows you to specify bit widths dynamically by using data register; for example, you can write BFINS D0,**1234{D3:D4}**.

Consider a typical 68020 instruction, the *bit field insert operation*, BFINS Dn,**<ea>{offset:width}**, that copies the bit field in register Dn to memory.

Consider BFINS D0,**1234{11:10}**. The least-significant 10 bits in register D0 are copied into the main store, starting at 11 bits (i.e., the offset) from bit 7 of the base byte address 1234 (Figure 4.19 has the same offset:width values which should help visualize this operation).

FIGURE 4.19 The 68020's bit field organization

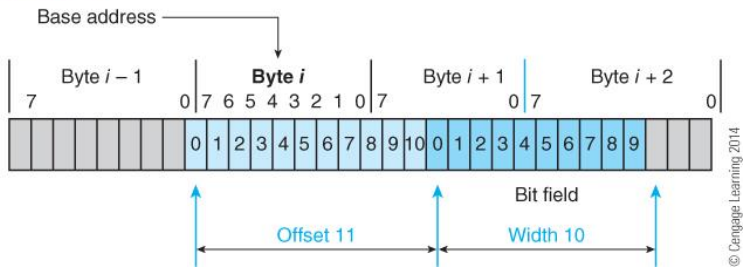
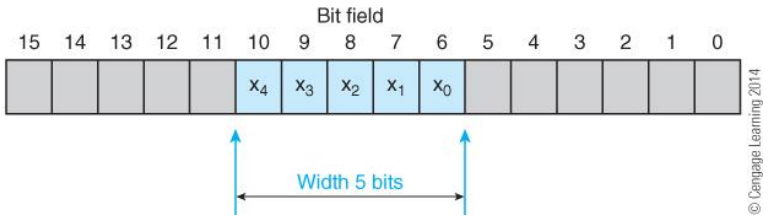


Figure 4.20 demonstrates a 5-bit data element X packed in a 16-bit word.

Suppose we wish to extract this bit field. Without bit field operations, we would typically load the data into a register, shift the data right to put it in the least-significant bit position and then clear the remaining bits to zero;.

```
MOVE  PQRS,D0                  ;Get the 16 bits of packed data into D0
LSR   #6,D0                    ;right-justify the bit field into D0–D5
AND   #%0000000000011111,D0    ;Clear all other bits of D0.
```

FIGURE 4.20 Packed data

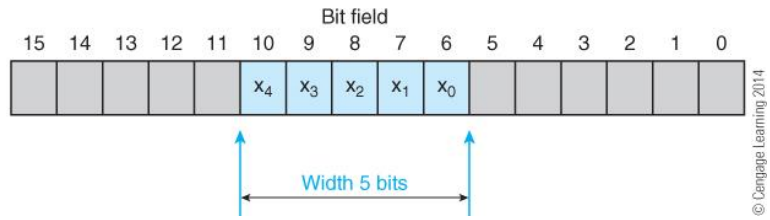


The 68020's *bit field extract* instruction, BFEXTU, performs this operation in one instruction:

```
BFEXTU  PQRS{5:5},D0        Get the packed data
```

The bit field offset is 5 because the position of the bit field is measured from the most significant bit of the base byte (i.e., bit 15 of the word). The first bit of the bit field is bit x_4 that is five bits to the right of bit 15.

FIGURE 4.20 Packed data



Bit field operations allow you to read a bit field from memory, to insert a bit field in memory, to clear/set/toggle all the bits of a bit field, and to test a bit field. Figure 4.21 demonstrates how the four-bit bit field in bits 6 to 3 of memory location 1000 can be moved to bits 4 to 1 of memory location in two instructions by

BFEXTU $1000,**{1:4},D0** Read the source bit field into D0
BFINS D0,**$1003,{3:4}** Store the bit field in memory

FIGURE 4.21 Using bit field instruction to move bits

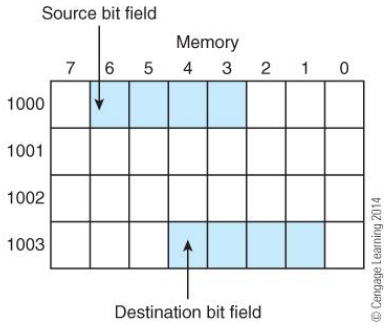
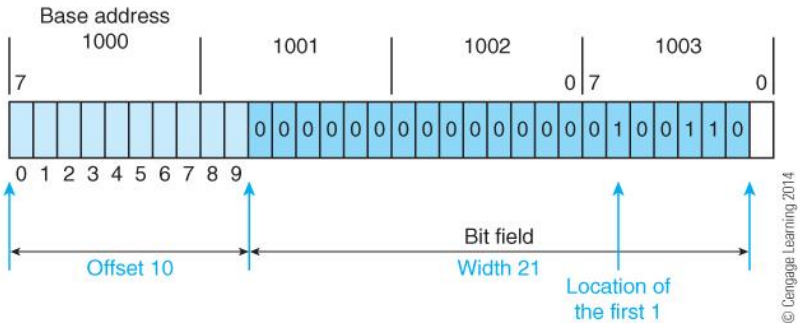


Figure 4.22 demonstrates a BFFFO with a 21-bit bit field beginning in byte 1001. We wish to locate the position of the **first bit set to a 1** within this bit field. If $1000 is the base byte, BFFFO $1000{10:21},**D0** scans the bit field , determines the location of the first 1 (i.e., bit 15 of the bit field) and loads 25 into register D1. The value is 25 because it's the location of the first 1 in the field plus the offset 10.

FIGURE 4.22 The BFFFO instruction



The Loop Counter

Here's another example of a special instruction from the 68K. The *decrement and branch instruction*, DBRA, allows you to specify one of eight loop counters, and one of *two* exit points. The loop can be terminated when the loop count has been exhausted or when a specific condition had been detected.

This instruction terminates the loop on a count of -1 rather than 0. The following fragment of code demonstrates how a DBCS (decrement and branch on carry set) is used to add together ten numbers but terminate if integer overflow occurs.

```
        MOVE  #10,D0        ;Set up loop counter ready to count down
        CLR   D1            ;Clear the total in register D1
        LEA   Table,A0      ;Point to the list of numbers
Next    ADD  (A0)+,D1       ;REPEAT: Add in the next number
        DBCS  D0,Next       ;UNTIL all added OR overflow
```

Without the DBCS instruction, the body of the loop would require four instructions.

MEMORY INDIRECT ADDRESSING

Memory indirect addressing lets you implement complex data structures.

Recall that register indirect addressing uses a pointer to access the required operand.

In *memory indirect addressing*, a register provides a pointer to *a pointer in memory*.

The actual operand is accessed by reading this *second* pointer and accessing the element at the address given by the pointer.

Four memory/register accesses are required; read the instruction, read the register containing the pointer to memory, read the memory containing the pointer to the operand, and access the operand.

Figure 4.23 illustrates *memory indirect addressing*, where a pointer register contains the 32-bit value 1234_{16}.

The contents of the target address specified by this pointer are 122488_{16}, and are used as a second pointer to access the actual operand.

If the initial pointer register is R1, the destination register is R2, and the instruction is a move, we can express this operation in RTL as

[R2] ← [[[R1]]]

FIGURE 4.23 Memory indirect addressing

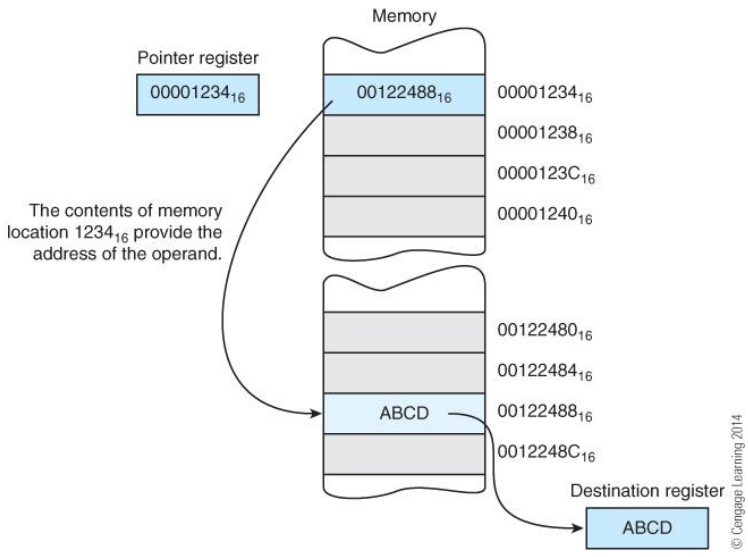
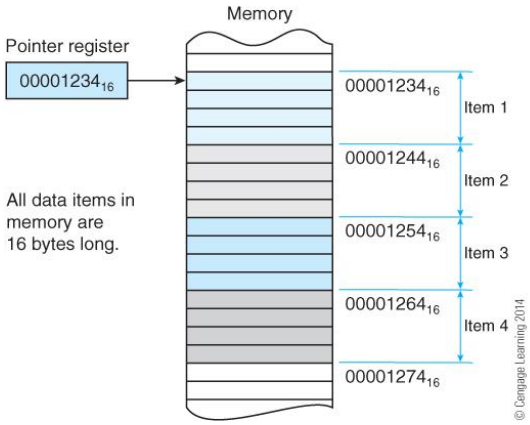


Figure 4.24 demonstrates a data structure consisting of consecutive 16-bytes values.

The pointer register contains 1234_{16}, corresponding to the first item in the structure.

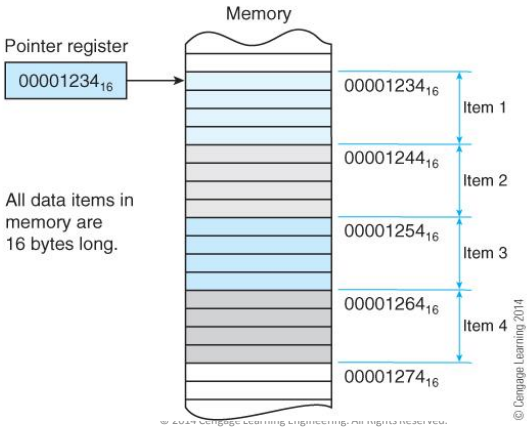
FIGURE 4.24 Accessing a *regular* data structure with register indirect addressing



To access item 2, you have to add 16 to the value in the pointer register.

Processors use a *register indirect with offset mode* to add a constant to a pointer; for example, the ARM uses LDR **r1**,[r0,#16] and the 68K MOVE (16,A0),**D1**.

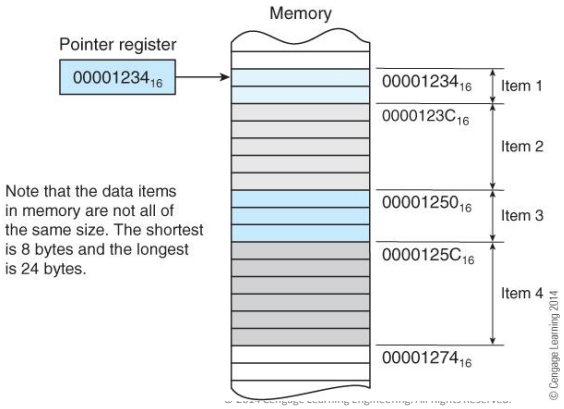
FIGURE 4.24 Accessing a *regular* data structure with register indirect addressing



Not all data structures are as well-ordered as that in Figure 4.24 where the size of each of the data items is the same.

Figure 4.25 illustrates the situation where each of the four items has a different size. We can't step through this data structure item-by-item just by adding a constant to the pointer register.

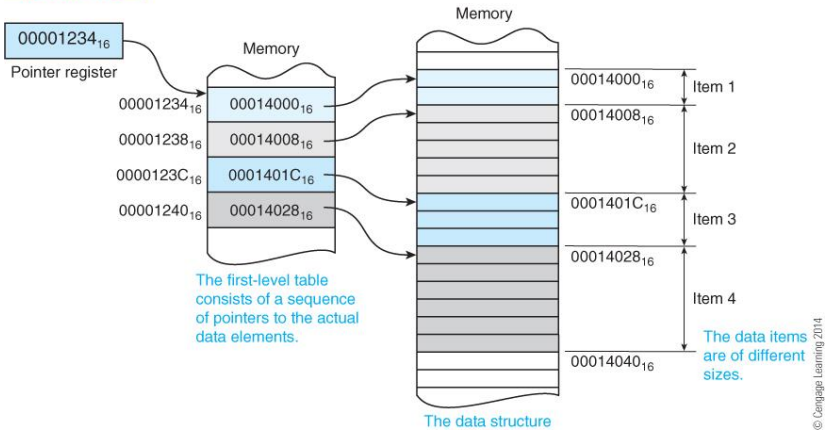
FIGURE 4.25 Accessing an irregular data structure



We can take an alternative approach to accessing data in structures with records of varying length.

Figure 4.26 uses a pointer register that points to a *table of pointers*.

FIGURE 4.26 Accessing irregular data structures via memory indirect addressing



Each of the pointers points to the actual record in memory.

You can step through the data items simply by incrementing the base pointer by four, because the base pointer steps through the table of pointers.

FIGURE 4.26 Accessing irregular data structures via memory indirect addressing

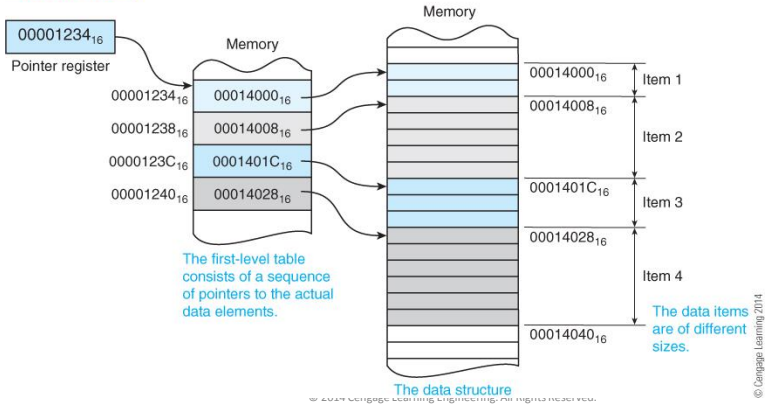
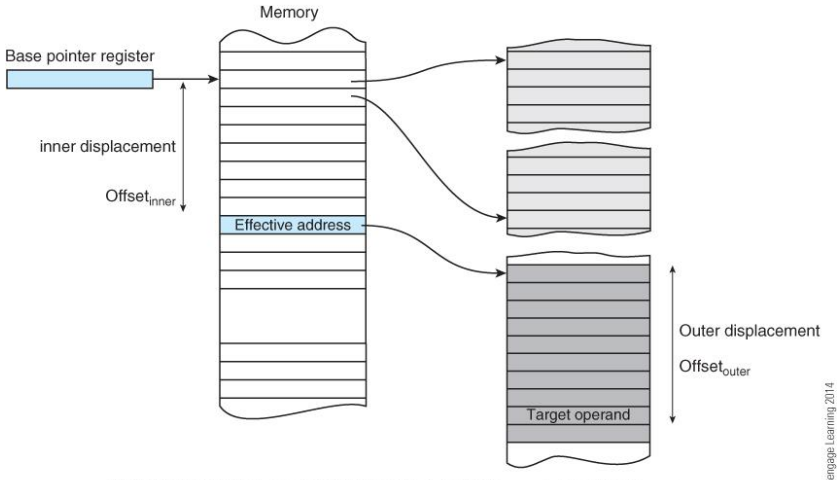


FIGURE 4.27 Accessing an operand within the target data structure



Target operand = [[[BasePointer] + $Offset_{inner}$] + $Offset_{outer}$]

The SWITCH construct

A common construct in many high-level languages is the switch that allows you to invoke one of *n* functions depending on the value of a variable. Suppose you were constructing a CPU simulator.

You might have an inner interpreter that looks something like the following to select one of four cases.

```
Switch (operation)
{
 case LOAD:      { LOAD  code; break:}
 case STORE:     { STORE code; break:}
 case ADD :      { ADD   code; break:}
 case BEQ :      { BEQ   code; break:}
}
```


Figure 4.28 illustrates a possible data structure for this construct where a table in memory holds pointers to the functions.

The required function is executed by loading the appropriate pointer into the program counter.

Let's implement a switch construct using a conventional CISC architecture such as the 68K.

We can use memory indirect addressing to call the required subroutine by executing

```
JSR ([A0,D0*4])      ;Call the subroutine specified by D0 (Table base in A0)
```

