Lecture 14: Subroutine- The Stack and Parameter passing

A subroutine is designed to carry out some particular function. In order to do this, it is almost always necessary to transfer data between the calling program and the subroutine. Up to now we have passed data to and from the subroutine via data registers. In this lecture we will see other means of passing parameters between a subroutine and the calling program.
In addition, while a program is running, it has access to memory locations, registers, peripherals, etc. The set of all these with their contents, is called the program's 'context.' A running program could be characterised at any instant by its context and its program code. We will discuss the importance of the program context in designing 'well behaved' subroutines.

Learning Outcomes:
On completion of this lecture, you will be able to:

- Distinguish different mechanisms for passing parameters between subroutines and calling programs;
- Discuss the advantage of re-entrancy;
- Discuss the notions of program context, processes and threads;
- Write well behaved subroutines that complete well defined tasks;
- Design memory efficient subroutines;

14.1 Parameter Passing

In a previous example, we called a subroutine SUBRUT to translate a signed binary word integer to a NUL-terminated sequence of BCD (Binary Coded Decimal) ASCII characters. When SUBRUT is invoked by the operation JSR SUBRUT, a jump is made to the entry point of the subroutine. This subroutine reads the contents of data register D0 and a return to the calling point is made with the BCD ASCII code of the number saved to the memory location point at by address register A0. As only a single word integer is passed from the calling program to the subroutine, a data register provides a handy vehicle to transfer the information. Unfortunately, you can't use registers to transfer large quantities of data to and from subroutines, owing to the limited number of registers.

14.1.1 Passing parameters on the stack
An ideal way of passing information between a subroutine and its calling program is via the stack. Suppose two 16-bit parameters, P1 and P2, are needed by a subroutine. The parameters are pushed on the stack immediately before the subroutine call:

```
MOVE.W      P1,-(A7) Push the first parameter
MOVE.W      P2,-(A7) Push the second parameter
```

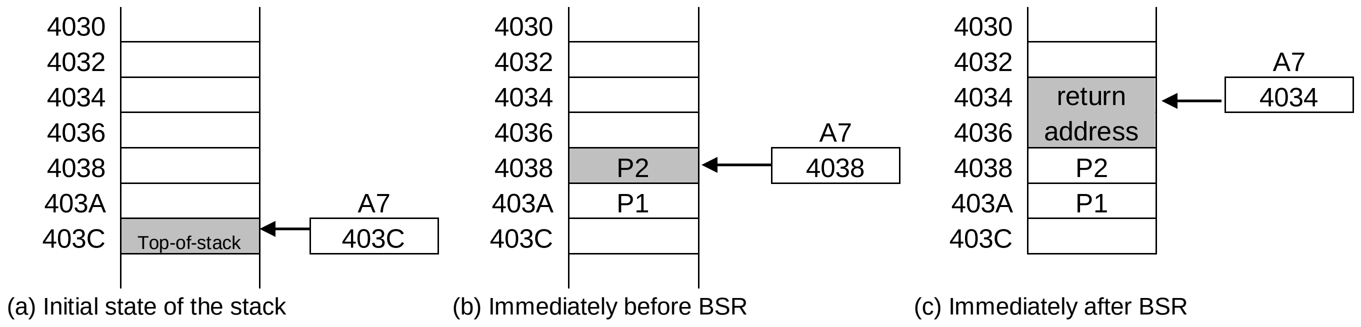
4030
4032
4034
4036
4038
403A
403C
Top-of-stack
A7
403C
(a) Initial state of the stack
P2
P1
A7
4038
(b) Immediately before BSR
return address
A7
4034
(c) Immediately after BSR

Fig. 14.1: Passing parameters on the stack

On entering the subroutine, the parameters can be retrieved from the stack by making a copy of it in another address register with *LEA (A7),A0*. This mechanism **avoids moving the system stack pointer A7 below the top-of-stack.** Now you can use A0 to get the parameters; for example, P1 can be loaded into D1 by *MOVE 6(A0),D1*. The offset 6 is required because the parameter P1 is buried under the return address (4 bytes) and P2 (2 bytes). Similarly, P2 can be loaded into D2 by *MOVE.W 4(A0),D2.*
New addressing mode: **'Address Register Indirect with displacement'** -- Look like:d(An)
The effective address (EA) is calculated by adding the contents of the specified address register, A0, to the sign-extended 16-bit displacement *d*: EA= d+[An].

By using the stack to pass parameters to a subroutine, the subroutine may be interrupted and then used by the interrupting program without the parameters being corrupted. As data is stored on the stack, it is not overwritten when subroutine is interrupted because new data is added at the top of the stack, and then removed after the interrupt has been serviced.

14.1.2 Passing parameters by value
In the previous example, we passed a copy of a parameter to the subroutine by pushing its value on the stack. This mechanism is called passing parameters by value. There are then two copies of the parameter: the original in the calling program and its copy on the stack. If a parameter is passed by value, changing it within the subroutine does not change its value in the calling program.

14.1.3 Passing parameters by reference
You can also pass a parameter to a subroutine by reference by passing its address on the stack. In this case, there is only one copy of the parameter.
We can now introduce a new instruction, push effective address, **PEA**, which pushes an address in the stack; for example, the operation *PEA PQR* pushes the address PQR on the stack. This instruction is equivalent to:

```
        MOVE.L        #PQR,-(A7)
```

In practice, you would pass parameter that are not changed in the subroutine by value, and only pass parameters that are to be changed by reference.

14.2 Re-entrant Subroutines

A subroutine that can be safely called while it's already executing is said to be *re-entran*t. Re-entrancy is possible as long as the re-use of the subroutine saves the contents of the registers employed to transfer data before it is re-used.

Why is re-entrancy a desirable feature in a computer? More importantly, why should a program wish to borrow a subroutine that is being used by another program? In a multitasking system, several programs or tasks run simultaneously. To be more precise, the operating system switches between user tasks so rapidly that to an outsider (e.g., a human operator) they appear to be running simultaneously. Imagine a situation in which task A is running and using subroutine X to perform a certain function. Suppose the operating system switches tasks while A is still in the middle of subroutine X. Task B also uses subroutine X. When B has finished with subroutine X, it must leave the subroutine in exactly the same condition it found X. Otherwise, the multitasking system simply would not work.

14.3 Designing 'Ecological' subroutines

Processes and Threads

• A saved context and program code is almost like a freeze dried program—just add ~~water~~ a processor, and the process comes to life.
• The combination of context and code is sometimes called a 'process.' A process can be running or stopped.
• Part of the job of an operating system is to schedule processes to give them time on a processor.

Well Behaved Subroutines

• Subroutines run in the same context as their callers—same register values, memory locations etc.
• Thus, subroutines are part of the caller's process. They are not independent of the caller.
• Care must be taken to ensure that subroutines do not damage the context of their callers. In a kind of shorthand, use of subroutines must be, eh, 'ecological' with respect to the environment.

'Ecological' Subroutines—Easy To Use

• Subroutines should be easy to understand!
• Ideally, subroutines are like building blocks, each just implementing one functionality.
• Subroutines should do *exactly and only* what is specified and *absolutely no more.* It is a serious error for a subroutine to do more than specified. Extra features frequently have unintended consequences.
• The scheme for passing parameters should be simple and neat.

'Ecological' Subroutines—Register Use

• If a subroutine uses a register for something, there's a chance it's already in use by the caller, so its contents may be important.
• Therefore, its contents must be saved before use, and then restored afterwards.
• In that way, the register's contents are preserved for the caller.
• Can be a bit costly!

'Ecological' Subroutines—Suggested Rules

• Use registers to pass parameters to and from a subroutine, where possible. Use 'parameter blocks' where possible. Use the stack if necessary.
• Save register contents to the stack before using the registers:
• That way, you can restore their original contents before returning to the caller.
• If a subroutine needs memory space, take it from the stack, if possible.
• Make sure all uses of the stack are 'balanced,' i.e. pushes=pops.

'Ecological' Subroutines—Memory Use

• Imagine a computer with two processes, both calling the same subroutine at the 'same' time.
• if the subroutine is written to always use the same memory locations to save private data, then
• Both copies of the subroutine could write their own private data to the same memory locations at the same time, causing severe problems. Such a subroutine would not be 're-entrant'
• Therefore, a subroutine should avoid using fixed memory locations for anything.

14.4 Conclusion:

This lecture provided you with general rules and guidance for writing efficient and flexible subroutines. Remember that a well designed subroutine should:
- only complete the actions specified in its requirements;
- preserve registers contents;
- Use memory space from the stack, is possible;
- be “re-entrant”.

REFERENCES

- Clements; Subroutines and Parameters, In: 68000 Family Assembly Language; pp.276-325; PWS Publishing Company; 1994.
- Dr. Mike Brady, Microprocessor Systems 1, dept of Computer Science, Trinity College Dublin: http://www.tcd.ie/Engineering/Courses/BAI/JS_Subjects/3D1/.
- Look on the Web. Look at http://www.mee.tcd.ie/~assambc/3D1.