

ELEC6027: VLSI Design Project  
Part 1: Microprocessor Research  
Topic: Subroutines

Ashley Robinson

Team: R4

Course Tutor: Mr B. Iain McNally

18<sup>th</sup> February, 2014

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Research</b>	<b>2</b>
2.1	Subroutine Context Save . . . . .	2
2.2	Operation of Stack Frames . . . . .	3
<b>3</b>	<b>Case Studies</b>	<b>4</b>
3.1	Intel 8086 . . . . .	4
3.2	ARM7TDMI (using Arm Thumb) . . . . .	6
<b>4</b>	<b>Conclusion</b>	<b>7</b>
	<b>References</b>	<b>8</b>
	<b>Bibliography</b>	<b>9</b>

# 1 Introduction

Subroutines, also known as procedures, methods, functions or just routines, are smaller sections of code inside larger programs designed to perform certain tasks as described in [3]. The motivation for subroutines is to produce code which is more efficient in size, easy to adapt and above all else maintain. They help form the foundations of third generation programming languages.

This report approaches the implication of hardware design by considering microprocessor architecture and targeted assembler subroutines. Designing hardware such that it is capable of executing subroutines only requires available memory and access to the current position in the program; usually a register called the Program Counter (PC). Designing hardware to call and return from subroutines efficiently can vastly improve program performance.

## 2 Research

### 2.1 Subroutine Context Save

Context save allows a microprocessor to switch execution focus but retain data such that the previous focus can be fully restored. Using call and return instructions when running a subroutine may automatically save context, usually the program counter, but may also require additional instructions to guarantee safe execution. Nested and recursive subroutines, introduced in [5], require dealing with multiple context saves while retaining the data for all previous calls. Listing 1 holds subroutine written in C that recursively calls itself to compute the factorial of a number. Such applications require context save support that permits many nested subroutines.

```
int factorial(int n){
    return n*factorial(n-1);
}
```

Listing 1: Factorial subroutine

A stack, as discussed in [6], is a Last-In-First-Out (LIFO) data structure which is used as store when the immediately accessible registers do not provide enough memory. The stack can be thought of to grow in size where a register called a Stack Pointer (SP) holds the address of the top most element in main memory. Two operations can be performed on the stack; *push*, which adds an item to the stack and increments the stack pointer, together with *pull*, which performs the reverse therefore shrinking the stack. It is convention for the stack to occupy main memory and grow down from the top

address towards memory allocated for other purposes; this can be seen in figure 1. Data can be accessed indirectly on the stack using standard load and store instructions. If the top of the stack is known then relative addressing can be used to go back and forth without having to change the SP.

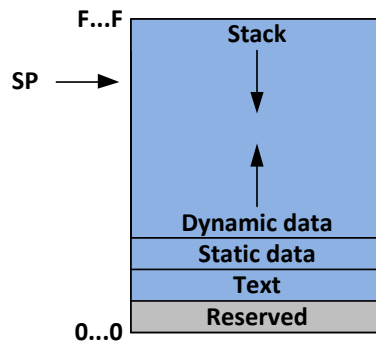


Figure 1: Allocation of the stack in main memory. Based on a MIPS architecture adapted from [6].

Context save can also be handled by the assembly code running on the processor. Function prologues and epilogues, considered in [11, 2], are used to move data. These follow conventions which usually advice the programmer on what register to place on the stack and how to do so from the caller and callee.

## 2.2 Operation of Stack Frames

A stack frame is a method of ordering data used when a subroutine is called. It is created by both the caller and the callee. It holds the paramters passed to the subroutine, the return address which should pointer at the caller and local variables created inside the callee. Figure 2 is a call stack; a stack of stack frames created by nesting subroutines. This is generally the case for most programs but optimisation can be used to improve memory allocation. Explained in [10], overlapping can be used to pas local variables on to a nested subroutine. Data returned from a function can be accessed using load instructions with addresses previously occupied by the stack.

Support for stack frames may be provided by a base pointer (BP). This usually takes on the value of the SP when the return address is placed on the stack. Therefore no calulations have to be done in order to destroy the local variable part of stack frame, just a simple overwrite.

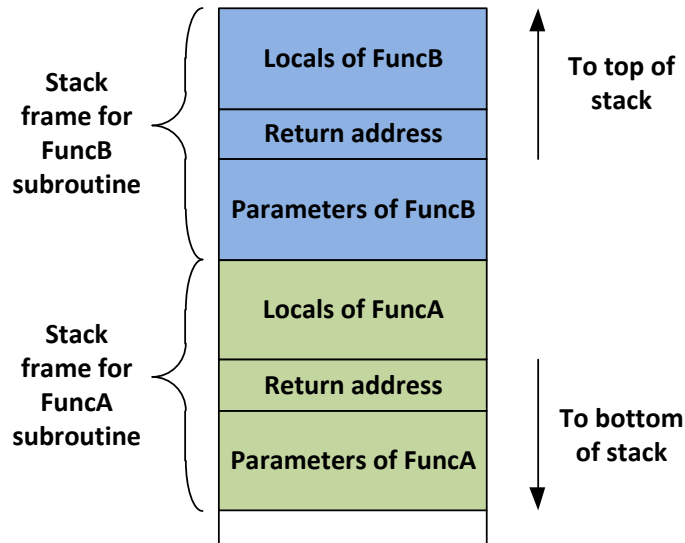


Figure 2: An example call stack built of two stack frames where *FuncA* is called first. Adapted from [10].

### 3 Case Studies

#### 3.1 Intel 8086

The Intel 8086 has support for procedure call instructions which automatically use the stack to organise data. The assembler held in listing 2 and 3 is written for the Intel 8086 microprocessor. This is a basic example of how stack frames are built to pass parameters to and from a subroutine. The main program in listing 2 loads two immediate values into registers then begins building a stack frame by pushing them to the stack. The subroutine is called to act upon the arguments passed via the stack. When control is passed back to the caller these set of instructions and the return value is extracted by using relative addressing from the BP then finally two stack pops completely destroy the stack frame. It is also possible to destroy the stack frame by using an add instruction on the SP shrinking it by one entry for addition of two.

```

main:      ; Main loop
          MOV    bp,sp      ; Init base ptr
          MOV    ax,42      ; Load arg1
          MOV    bx,69      ; Load arg2
          PUSH   bx         ; Push arg1 to stack
          PUSH   ax         ; Push arg2 to stack
          CALL   adder      ; Call the subroutine
          MOV    cx,[bp-12] ; Access return value
          POP    ax         ; Restore all registers
          POP    bx
          JMP    main

```

Listing 2: 8086Caller.asm

When the subroutine, in listing 3, is called the return address is pushed onto the stack. This built-in support for the stack handles branching and next line address storage using a call function. To start the BP is placed on the stack so the SP has restoration value. Reducing the value of the SP allocates space for local variables. The first argument is placed in memory as a local variable; this is unnecessary but serves as example. The second argument is loaded into a working register. The first local variable is added to the working register which is then placed in to the memory as the second local variable. Finally the SP and the BP are restored and a return instruction hands control over to the caller. This is all part of the calling convention for subroutines using stack frames on the 8086 microprocessor [1].

```

adder PROC ; Subroutine
          PUSH   bp         ; Push base ptr to stack
          MOV    bp,sp      ; Set base ptr to stack ptr
          SUB     sp,4       ; Allocate local variable space (2 ints)
          MOV    ax,[bp+4]   ; Load arg1 into Working
          MOV    [bp-2],ax   ; Load arg1 into Local1
          MOV    ax,[bp+6]   ; Load arg2 into Working
          ADD     ax,[bp-2]   ; Add to contents of working reg
          MOV    [bp-4],ax   ; Write Local2 with result
          MOV    sp,bp      ; Return stack ptr
          POP    bp         ; Restore base ptr
          RET             ; Done
adder ENDP

```

Listing 3: 8086Callee.asm

This code was tested upon an 8086 emulator [9]. The emulator provides a complete overview of the flow of data within the processor including the stack. Figure 3a shows the emulator during the execution of the subroutine just before the SP is overwritten with the BP. Figure 3b is an abstraction of the stack with data labels corresponding to the subroutine.

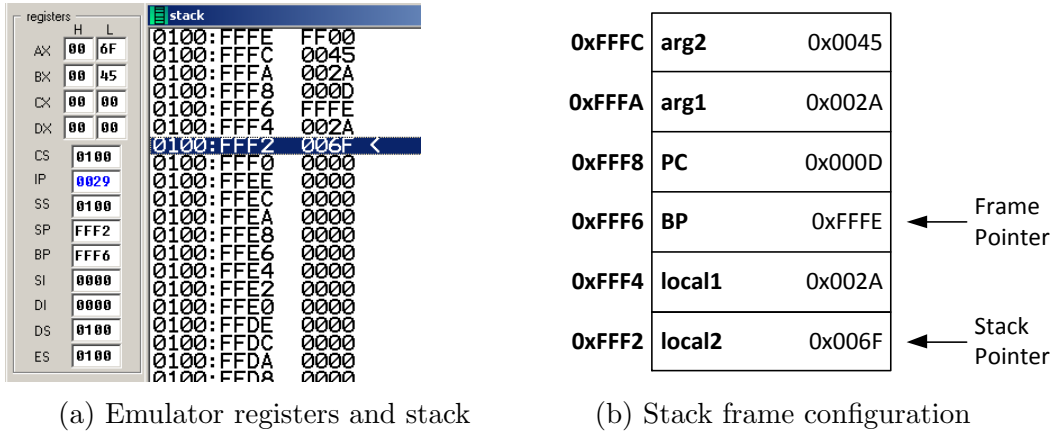


Figure 3: 8086 stack operation

### 3.2 ARM7TDMI (using Arm Thumb)

The ARM7TDMI is a 32-bit RISC microprocessor with an emphasis on low-power design and pipelining for high throughput which is detailed in [7]. It has two instruction sets. One of which is Arm Thumb, explained in [8], a low density compressed 16-bit subset of the ARM assembly language where the benefits are considered in [4]. It most notably assisted ARM into moving into the mobile phone market by reducing required program memory. A user selectable flag is set to switch between instruction sets.

This architecture does not have built-in support for calling subroutines using the stack. When the branch instruction is used, as seen in listing 4, the program counter is overwritten with the address of the corresponding label. The address of the next line of code, which should be returned to after the subroutine, is placed into the link register. Calling conventions suggests leaving this register untouched and simply moving the data back into the program counter on a return.

main	MOV	r0, #42	; Load arg1
	MOV	r1, #69	; Load arg2
	PUSH	r0	; Push arg1 to stack
	PUSH	r1	; Push arg2 to stack
	BL	adder	; Branch to subroutine
	POP	r0	; This line is held in the link register
	POP	r0	; Result pop from arg1 spot
	BL	main	

Listing 4: ArmCaller.asm

In this case the link register is pushed onto the stack from the subroutine therefore requiring the subroutine to pop the value into the program counter

in order to return. This is an example of important assembler context save. If the prologue and epilogue are not respected then a call stack is never built therefore no nested functions can be used.

Listing 5 holds the subroutine and handles placing the return address on the stack. Relative addressing on the stack is required to draw the two arguments out and replace the first with the output of the function. Although this subroutine performs the same task as code held in listing 3 only one local variable is used as it is never really required.

adder	PUSH	lr	; Link register holds return address
	SUB	sp, sp, #4	; Make space for locals on the stack
	LDR	r0, [sp, #12]	; Get arg1 off stack
	LDR	r1, [sp, #8]	; Get arg2 off stack
	ADD	r0, r0, r1	; Do the add
	STR	r0, [sp, #0]	; Store the add as a local
	ADD	sp, sp, #4	; Destroy local variable space
	POP	pc	; Restore program counter and return

Listing 5: ArmCallee.asm

## 4 Conclusion

It is clear that whether a microprocessor has built-in support for a stack or not the ability to call subroutines is unaffected. The support does make calling more efficient and easier to code.



## References

- [1] James Archibald. The c calling convention and the 8086: Using the stack frame. <http://ece425web.groups.et.byu.net/stable/labs/StackFrame.html>, 2013. Online. Accessed Feb 2014.
- [2] Pollard J. B. The gen on function perilogues. <http://homepage.ntlworld.com/~jonathan.deboynepollard/FGA/function-perilogues.html>, 2010. Online, Accessed Feb 2014.
- [3] Allison June Barlow Chaney. Subroutine. <https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Subroutine.html>, Oct 2013. Online. Accessed Feb 2014.
- [4] Hugue M. Don't we all need arms? <http://www.cs.umd.edu/class/fall2001/cmsc411/proj01/arm/>, 2001. Online, Accessed Feb 2014.
- [5] B. I. McNally. Vlsi group design project: Programs. <https://secure.ecs.soton.ac.uk/notes/bim/notes/fcde/assign/programs.html>, Jan 2012. Online, Accessed Feb 2014.
- [6] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 4th edition, 2008. pp114–121.
- [7] ARM Holdings plc. Arm7tdmi data sheet. <http://www.ndsretro.com/download/ARM7TDMI.pdf>, Aug 1995. Online. Accessed Feb 2014.
- [8] ARM Holdings plc. Thumb instruction set quick reference card. [http://www.eng.auburn.edu/~nelson/courses/elec5260\\_6260/Thumb%20Instructions.pdf](http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/Thumb%20Instructions.pdf), Oct 2003. Online. Accessed Feb 2014.
- [9] Daniel B. Sedory, Randall Hyde, Eric Isaacson, Barry Allyn, Tomasz Grysztar, Saul Coval, Bob Brodt, Jordan Russell, and Jeremy Gordonii. emu8086. <http://www.emu8086.com/>, 2013. Online. Accessed Feb 2014.
- [10] Wikipedia. Call stack. [http://en.wikipedia.org/wiki/Call\\_stack](http://en.wikipedia.org/wiki/Call_stack). Online, Accessed Feb 2014.
- [11] Wikipedia. Function prologue. [http://en.wikipedia.org/wiki/Function\\_prologue](http://en.wikipedia.org/wiki/Function_prologue). Online, Accessed Feb 2014.

## Bibliography

- ARM, Infocenter, <http://infocenter.arm.com/help/index.jsp>, Online. Accessed Feb 2014.
  - A valuable resource for all things ARM.
- Lin C., Understanding the Stack, <http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/stack.html>, 2004, University of Maryland, Online. Accessed Feb 2014.
  - General stack operation.
- Mathur S, *Microprocessor 8086: Architecture, Programming and Interfacing*, PHI Learning Private Limited, 2011.
  - Intel 8086 microprocessor resource
- Patterson D A and Hennessy J L, *Computer Organisation and Design: The Hardware/Software Interface*, Morgan Kaufman, 4th Edition, 2009.
  - Lots of processor concepts and MIPS examples