Subroutines

Subroutines ဆိုတာ assembly program တွေမှာသုံးတဲ့ *function* ဖြစ်ပါတယ်။ *Procedure*, *routine* လို့လည်း ခေါ်ပါတယ်။ C program တွေမှာ function တွေကို အသုံးပြုသလိုပဲ assembly program တွေမှာ subroutine တွေကို အသုံးပြုပါတယ်။ Subroutine သုံးခြင်းရဲ့ အဓိက အားသာချက် (၂) ခုကတော့ ရှုပ်ထွေးတဲ့ problem တစ်ခုကို ပိုပြီးရိုးရှင်းတဲ့ subtask တွေအဖြစ် ဖြိုခွဲခြင်းအားဖြင့် debug လုပ်တာ၊ error ရှာတာ ပိုလွယ်ကူစေပြီး program တစ်ခုမှာ ရေးပြီးသား subtask တစ်ခုကို duplicate လုပ်စရာမလိုပဲ အကြိမ်ကြိမ် ပြန်ခေါ်သုံးလို့ရခြင်းကြောင့် အချိန်ကုန်လူပင်ပန်းသက်သာစေပါတယ်။

C language ရဲ့ *function* တွေလိုပဲ subroutine တစ်ခုက input arguments တချို့ယူပြီး result တစ်ခု return ပြန်ပါတယ်။ ဒါပေမယ့် subroutine တစ်ခုကိုရေးမယ်ဆိုလျှင် သတိထားလုပ်ဆောင်ရမယ့် အချက်တွေရှိပါတယ်။

(၁) Preserve and recover the caller's environment

subroutine တစ်ခုက caller နဲ့ဆက်စပ်သော register တွေကို မထိခိုက်၊ မဖျက်ဆီးရပါဘူး။

- အရင်ဆုံး subroutine စစချင်းမှာ caller ရဲ့ environment ဖြစ်တဲ့ register တွေအား (PUSH လုပ်ခြင်းဖြင့်) stack ထဲကို သိမ်းရပါတယ်။ အဲ့ဒါမှသာ caller ရဲ့ environment ကို မထိခိုက်မှာ ဖြစ်ပါတယ်။
- ပြီးမှပဲ subroutine အဆုံးမှာ stack မှာသိမ်းထားတဲ့ register တွေကို (POP လုပ်ပြီး) ထုတ်ရပါတယ်။

(၂) Subroutine အားလုံးက ARM Embedded application binary interface (EABI) က သတ်မှတ်ထားတဲ့ စံ တစ်ခုကို လိုက်နာရပါမယ်။ ထိုမှသာလျှင် assembly ဖြင့် ရေးထားတဲ့ subroutine တစ်ခုကို C program တစ်ခုကနေ ခေါ်သုံးလို့ရမှာဖြစ်သလို၊ programmer တစ်ယောက်နဲ့တစ်ယောက် code တွေ share တဲ့နေရာမှာလည်း ပိုအဆင်ပြေစေမှာ ဖြစ်ပါတယ်။

Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and Return value	No	If the 1st argument has 64 bits, r1:r0 hold it (r1 is upper word, r0 is bottom word). If the 2nd argument has 64 bits, r3:r2 hold it. If more than 4 arguments, use the stack. If the return has 64 bits, then r1:r0 hold it. If the return has 128 bits, then r0-r3 hold it.
r1	Argument 2	No	
r2	Argument 3	No	
r3	Argument 4	No	
r4	General-purpose V1	Yes	Variable registers for holding local variables.
r5	General-purpose V2	Yes	
r6	General-purpose V3	Yes	
r7	General-purpose V4	Yes	
r8	General-purpose V5	Yes	
r9	Platform specific/V6	No	Usage is platform-dependent.
r10	General-purpose V7	Yes	Variable registers for holding local variables.
r11	General-purpose V8	Yes	
r12 (IP)	Intra-procedure-call register	No	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Yes	SP must be the same after a subroutine has completed.
r14 (LR)	Link register	No	LR does not have to contain the same value after a subroutine has completed.
r15 (PC)	Program counter	N/A	Do not directly change PC.

Table: Standard of register usage of a subroutine

Subroutine တစ်ခုကို ခေါ်သုံးဖို့ branch and link (BL) instruction ကို သုံးရပါတယ်။ BL instruction က အလုပ် (၂) ခုလုပ်ပေးပါတယ်။

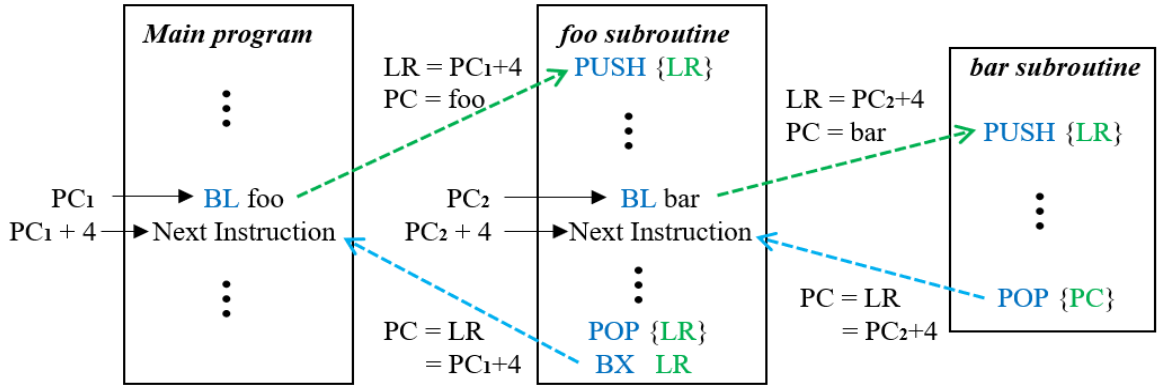
(၁) BL instruction ရဲ့ကပ်လိုက် instruction ရဲ့ memory address ကို link register (LR) ထဲကိုသိမ်းပါတယ်။ LR မှာရှိတဲ့ address ကို *return address* လို့လည်း ခေါ်ပါတယ်။ (LR ဆိုတာ register r14 ဖြစ်ပြီး သူ့အလုပ်ကတော့ subroutine တစ်ခု အလုပ်ပြီးသွားတဲ့အခါ caller program မှာ လုပ်ဆောင်ရမယ့် instruction ရဲ့ memory address ကို သိမ်းပါတယ်။)

(၂) ကိုယ်ခေါ်ထားတဲ့ subroutine ရဲ့ ပထမဆုံး instruction ရဲ့ memory address ကို program counter (PC) ထဲကို သိမ်းပေးပါတယ်။ (PC ဆိုတာ register r15 ဖြစ်ပြီး သူ့အလုပ်ကတော့ processor လုပ်ဆောင်တော့မယ့် next instruction ရဲ့ memory address ကို သိမ်းပါတယ်။)

Subroutine တစ်ခုကနေထွက်ဖို့ နည်းလမ်း (၂) ခုရှိပါတယ်။

(၁) Branch and exchange instruction "**BX LR**" ကိုအသုံးပြုခြင်း (သို့)

(၂) Link register (LR) တန်ဖိုးကို stack ထဲကနေဆွဲထုတ်ပြီး program counter (PC) ထဲကို သိမ်းခြင်း — "**POP {PC}**" (subroutine ထဲကနေ stack ထဲကို LR တန်ဖိုး အရင် သိမ်းဖို့တော့ လိုပါတယ် — "PUSH {LR}") ဆိုပြီး ရှိပါတယ်။



An example of calling subroutines

အပေါ်ကပုံကို ကြည့်ရအောင်။ ။ *Main* program ကနေ "BL foo" ဆိုပြီး *foo* function ကိုခေါ်တဲ့အခါမှာ processor ကနေပြီးတော့ (၁) $PC_1 + 4$ ကို LR ထဲထည့်၊ (၂) foo ရဲ့ memory address ကို PC ထဲကို သိမ်းပါတယ်။ ထို့နောက် *foo* ကနေ *bar* ကို ခေါ်တဲ့အခါ processor က $PC_2 + 4$ ကို LR ထဲ သိမ်းပြီး *bar* ရဲ့ ပထမဆုံး instruction address ကို PC ထဲကို ထည့်ပါတယ်။ *bar* ကနေ "POP {PC}" လုပ်တဲ့အခါ LR တန်ဖိုးက $PC_2 + 4$ ပြန်ဖြစ်သွားပါတယ်။ *foo* ကနေ "BX LR" သုံးပြီး ထွက်တဲ့အခါ processor က LR တန်ဖိုး (PC_1+4) ကို PC ထဲ ထည့်ပါတယ်။

Subroutine တစ်ခုက အခြား subroutine တစ်ခုကို ခေါ်သုံးရင် Link Register (LR) ကို သိမ်းထား (preserve) ဖို့ လိုပါတယ်။ တစ်ကယ်လို့ *foo* က preserve and recover (သိမ်း/ထုတ်) မလုပ်ဘူးဆိုရင် နောက်ဆုံးမှာရှိတဲ့ "BX LR" က *main* program ကို ပြန်ရောက်နိုင်မှာ မဟုတ်ပါဘူး။ ဘာလို့လဲဆိုတော့ processor က $PC_1 + 4$ အစား $PC_2 + 4$ ကို PC ထဲကို မှားယွင်းစွာ သိမ်းထားမိမှာ ဖြစ်ပါတယ်။ လိုချင်တဲ့ရလာဒ် ရနိုင်မှာမဟုတ်ဘူးလို့ ဆိုလိုတာပါ။

Factorial ရှာတဲ့ subroutine တစ်ခုကို ဥပမာကြည့်ရအောင်။

C program	Address	Assembly Program
		AREA main, CODE, READONLY
		EXPORT __main
int factorial (int n);		ENTRY
int main(void) {		__main PROC
factorial(5);	0x0800012E	MOV r0, #5 ; *factorial(5)*
while(1);	0x08000130	BL factorial
}	**0x08000134**	stop B stop
		ENDP
int factorial (int n) {		**factorial** PROC
int f;	0x08000136	PUSH {r4, LR} ; *preserve*
	0x08000138	MOV r4, r0 ; *r4 = n*
if(n == 1)	0x08000134	CMP r4, #1
	0x0800013C	BNE else ; *if n ≠ 1*
f = 1;	0x0800013E	MOV r0, #1 ; *f = 1*
	0x08000140	loop POP {r4, PC} ; *return*
else	0x08000142	else SUB r0, r4, #1 ; *n - 1*
f = n * factorial(n-1);	0x08000144	BL **factorial** ; *r0 is input*
	0x08000148	MUL r0, r4, r0 ; *n * f(n-1)*
return f;	0x0800014C	B loop
}		ENDP
		ALIGN
		END

Example: Calculating factorial number in C and assembly

အပေါ်ကဥပမာကတော့ recursive သုံးပြီး factorial ရှာတဲ့ program ဖြစ်ပါတယ်။ (အလယ်က memory address ကတော့ board တစ်ခုနဲ့တစ်ခု တူချင်မှတူမှာပါ။) *main* ကနေ *factorial* ကို "BL factorial" ဆိုပြီး ခေါ်လိုက်ပါတယ်။ အဲ့ချိန်မှာ LR က **0x08000134** ဖြစ်ပြီး၊ PC က 0x08000136 ဖြစ်မှာ ဖြစ်ပါတယ်။ *factorial* subroutine ထဲရောက်တော့ r4 နဲ့ LR ကို stack ထဲ အရင်သိမ်းပါတယ် "PUSH {r4, LR}"။ Input ဖြစ်တဲ့ r0 ဟာ 1 နဲ့မညီမချင်း "MUL r0, r4, r0" instruction ကို မသွားသေးဘဲ "BL factorial" အထိပဲ ပတ်နေမှာ ဖြစ်ပါတယ်။ အပေါ်မှာ ပြောခဲ့တဲ့အတိုင်း BL တစ်ခုချင်းစီက သူ့ရှေ့မှာရှိတဲ့ instruction ရဲ့ address ကို LR ထဲကို သိမ်းပါတယ်။ အဲဒါကြောင့် *factorial* subroutine ထဲကနေ ပထမဆုံး "BL factorial" ကိုသွားပြီးပြီဆိုတာနဲ့ LR တန်ဖိုးက 0x08000148 ("MUL r0, r4, r0") ဖြစ်နေမှာပါ။ ဒီအထိကို အောက်မှာပြထားတဲ့အတိုင်း stack ထဲမှာ သိမ်းထားမှာ ဖြစ်ပါတယ်။

Memory Address	Memory Content	
0x20000600		
0x200005FC	**0x08000134** (LR)	
0x200005F8	0 (r4)	[5]
0x200005F4	0x08000148 (LR)	
0x200005F0	5 (r4)	[4]
0x200005EC	0x08000148 (LR)	
0x200005E8	4 (r4)	[3]
0x200005E4	0x08000148 (LR)	
0x200005E0	3 (r4)	[2]
0x200005DC	0x08000148 (LR)	
0x200005D8	2 (r4)	[1]
0x200005D4	0x08000148 (LR)	
0x200005D0		

Table: Stack content immediately after factorial (1) completes.

r4 က 1 နဲ့ညီပြီဆိုတာနဲ့ else label ကို မသွားတော့ဘဲ r0 (သို့) input ကို 1 ထား return ပြန်ပြီး loop label ကို သွားတော့မှာ ဖြစ်ပါတယ်။ ဆိုချင်တာက r4 တန်ဖိုးကို stack ထဲကို မသိမ်းတော့တာပါ။ အခဲ့တော့ stack မှာ နောက်ဆုံးသိမ်းထားတဲ့ r4 ရဲ့တန်ဖိုးက 2 ပဲ ရှိနေမှာပါ။ loop label မှာ POP {r4, PC} ပဲရှိပါတယ်။ r4 နဲ့ PC ကို POP လိုက်တော့ r4 = 2, PC ကတော့ 0x08000148 နေရာကို ပြမှာဖြစ်ပါတယ်။ အခဲ့ဒါကြောင့် “MUL r0, r4, r0” ကို လုပ်ဆောင်တော့မှာ ဖြစ်ပါတယ်။

[1] အခဲ့နေရာမှာ **r0 = 1**, r4 = 2 ရှိနေပါတယ်။ ပထမ MUL အပြီးမှာတော့ r0 = 1*2 = 2 ရှိမှာပါ။ **[2]** ပြီးလျှင် loop label ကိုပြန်သွားပြီး POP ပြန်လုပ်မှာ ဖြစ်ပါတယ်။ ထို့ကြောင့် r4 = 3, PC = 0x08000148 ဖြစ်ပြီး MUL instruction ကိုပဲ လုပ်ဆောင်နေဦးမှာ ဖြစ်ပါတယ်။ MUL instruction အပြီးမှာတော့ r0 = 2*3 = 6 ဖြစ်မှာပါ။ **[3]** loop label ပြန်သွား၊ r4 = 4, PC = 0x08000148၊ MUL instruction လုပ်၊ r0 = 6*4 = 24။ **[4]** နောက်ထပ် loop label ပြန်သွား၊ r4 = 5, PC = 0x08000148၊ MUL instruction လုပ်၊ **r0 =** 24*5 = **120**။ **[5]** နောက်ဆုံးမှာတော့ loop label ပြန်သွားပြီး POP လိုက်တော့ r4 = 0 (or) original value, PC = **0x08000134** ပြန်ဖြစ်သွားပါပြီ။ Memory address **0x08000134** ဟာ *main* program ထဲက *stop* label ဖြစ်တာကြောင့် program က infinite loop ပတ်သွားမှာဖြစ်ပါတယ်။

Subroutine အကြောင်းကတော့ လိုတိုရှင်းဒီလောက်ပါပဲ။ နားမလည်တာရှိရင် မေးမြန်းနိုင်ပါတယ်။ အမှားပါရင်လည်း ဝေဖန်ထောက်ပြပေးကြပါခင်ဗျာ။ ။