CPE 221

Chapter 4 – Instruction Set Architecture Breadth and Depth

Dr. Rhonda Kay Gaede

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

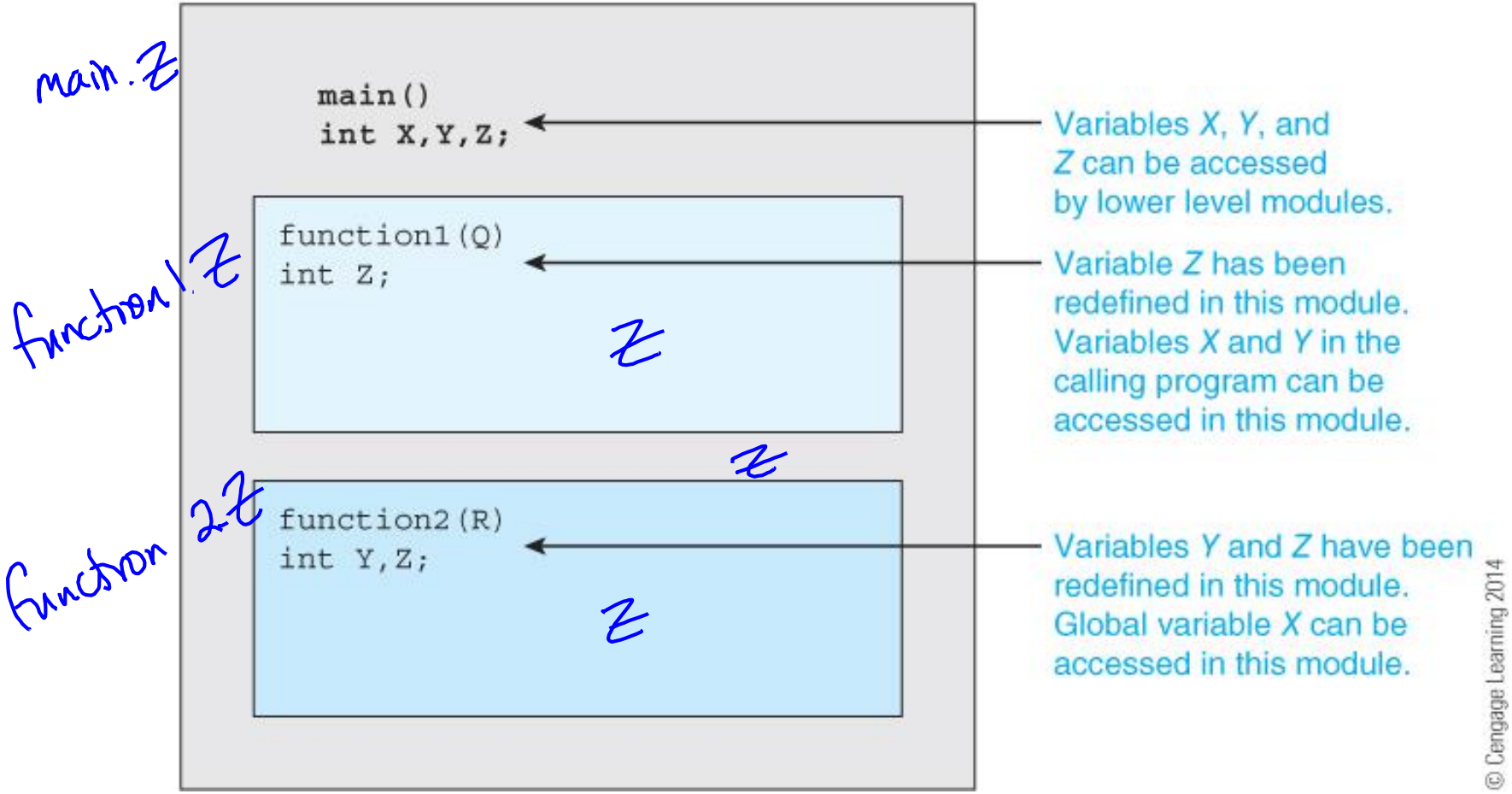
Electrical and Computer Engineering

4.1 Data Storage 101

- High-level language programmers use __variables__ to represent any type of __data__ __element__ defined by the programmer (e.g., __byte__, __string__, __struct__). array
- 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__.
- 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__.

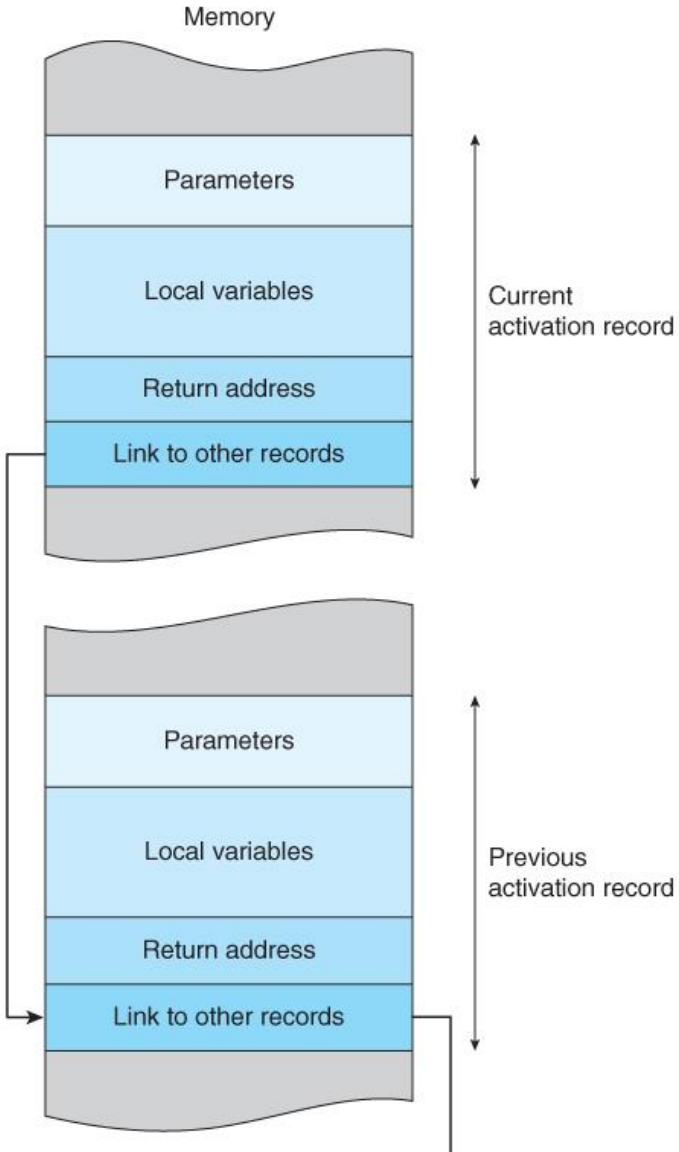
4.1 Illustration of Scope

FIGURE 4.1 The concept of scope



4.1 An Activation Record (Frame)

FIGURE 4.2 The activation record



- 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__.

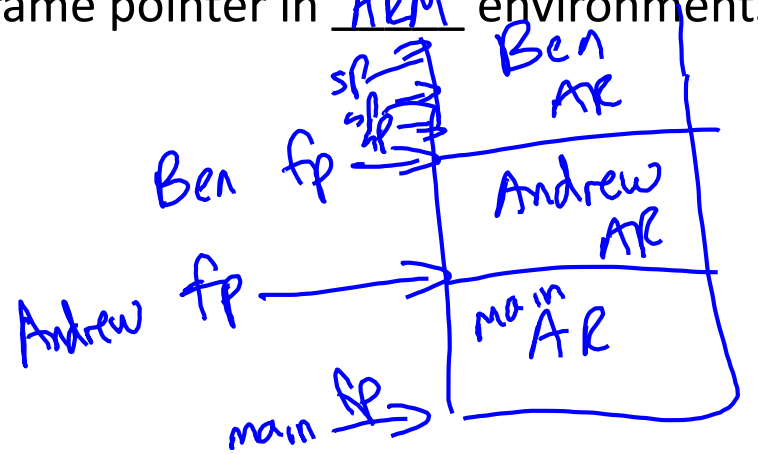
4.1 Stack Pointer and Frame Pointer

- 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.



4.1 Multiply_by_Adding with Subroutines

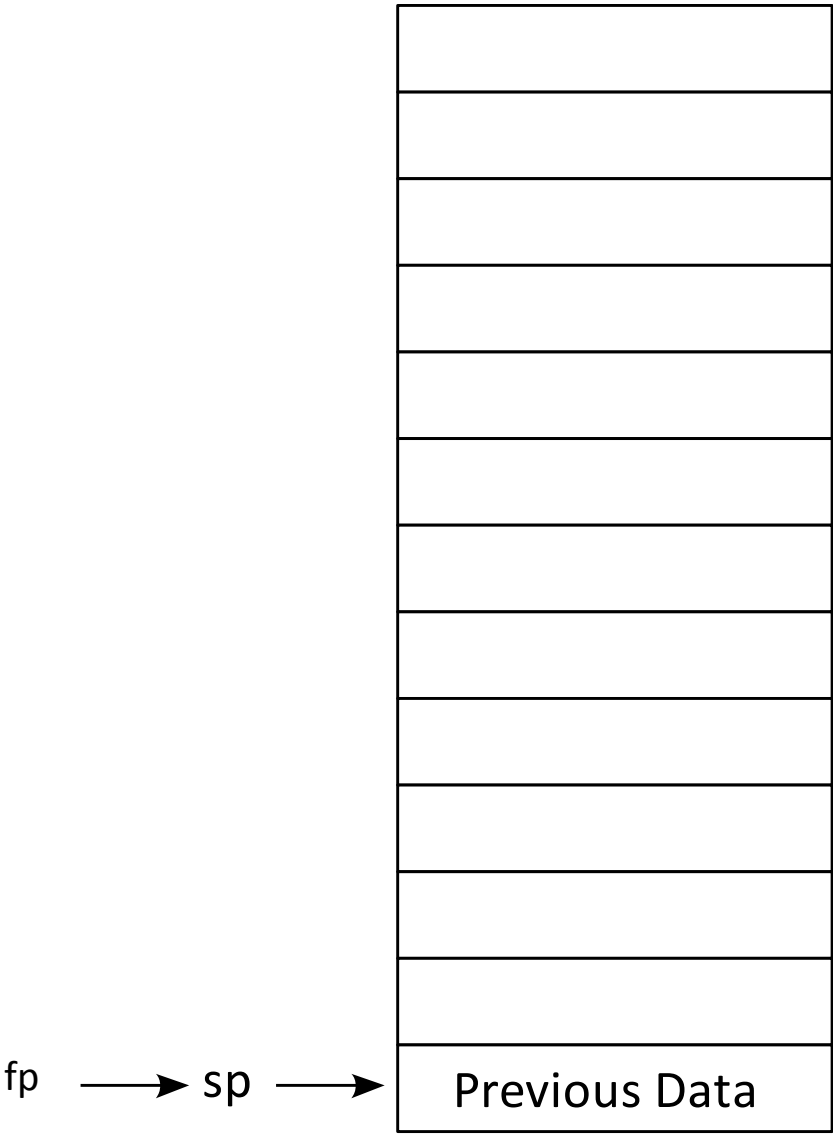
```
int mpy_ne(int, int)
int abs(int)

int main() {
  int first = 8;
  int second = -9;
  int result;
  result = mpy_ne(first, second);
}

int  mpy_ne (int num1; int num2){
  int a, b, mult;
  a = abs(num1);
  b = abs(num2);
  mult = 0;
  for (i = 0; i < a; i++)
    mult = mult + b;
  if (num1 < 0) mult = -mult;
  if (num2 < 0) mult = -mult;
  return mult;
}

int abs(int x){
  if (x < 0) x = -x;
  return x;
}
```

positive result

4.1 Multiply By Adding w Negative

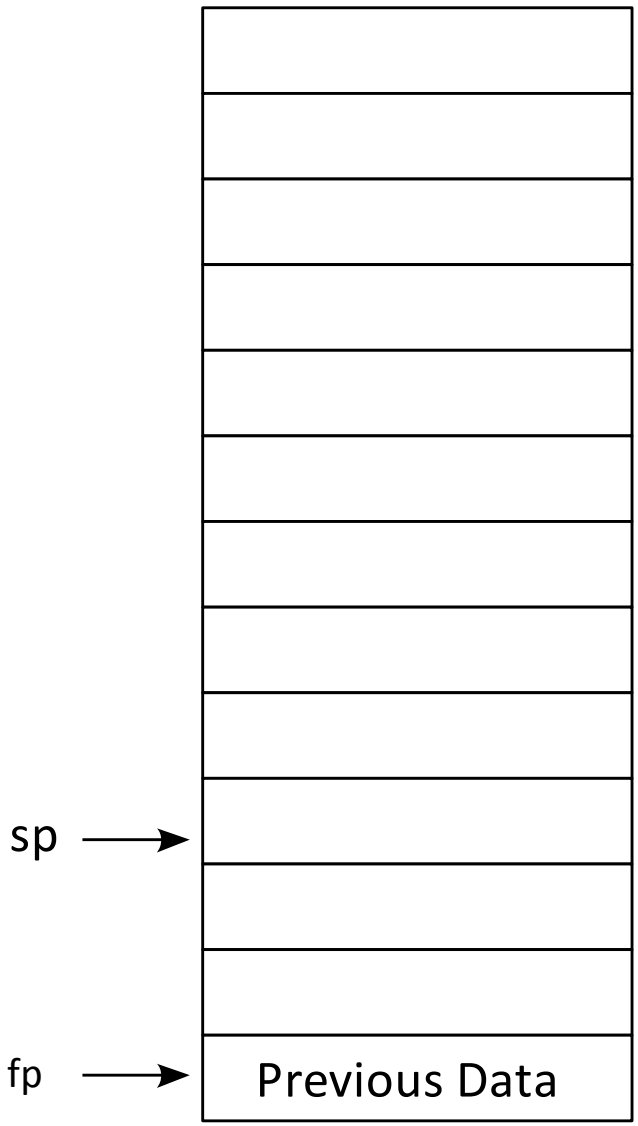
```
AREA MULTIPLY_BY_ADDING_WNEG, CODE, READONLY
        LDR   r1, num1                  ; Put num1 in r1.
        LDR   r2, num2                  ; Put num2 in r2.
        ADR   r9, result          ; Put &result in r9
        MOV   r3, #0              ; Set r3 to 0, it will hold the result.
        TEQ   r1, #0              ; Compare first num to 0
        BEQ   done                ; If first num is 0 done, result = 0.
        TEQ   r2, #0              ; Compare second numto 0
        BEQ   done                ; If second numis 0, done, result = 0.
        CMP   r1, #0
        RSBMI r4, r1, #0
        CMP   r2, #0
        RSBMI r5, r2, #0
adding  ADD   r3, r3, r5          ; Add num2.
        SUBS  r4, r4, #1          ; Decrement r4, the abs of num1.
        BEQ   adjust               ; If r4 = 0, done adding, go to adjust.
        B     adding               ; Otherwise, need to add again.
adjust  MOVS  r1, r1               ; Done adding, now adjust result.
        RSBMI r3, r3, #0          ; If num2 negative, negate result.
        MOVS  r2, r2
        RSBMI r3, r3, #0          ; If num1 negative, negate result.
done    STR   r3, [r9]
num1    DCD   -8                        ; Give num1 a value
num2    DCD   -9                        ; Give num2 a value
result  SPACE 4
        END
```

Multiply_by_Adding Stack Contents




```
16  MOV   fp, sp
```

Multiply_by_Adding Stack Contents



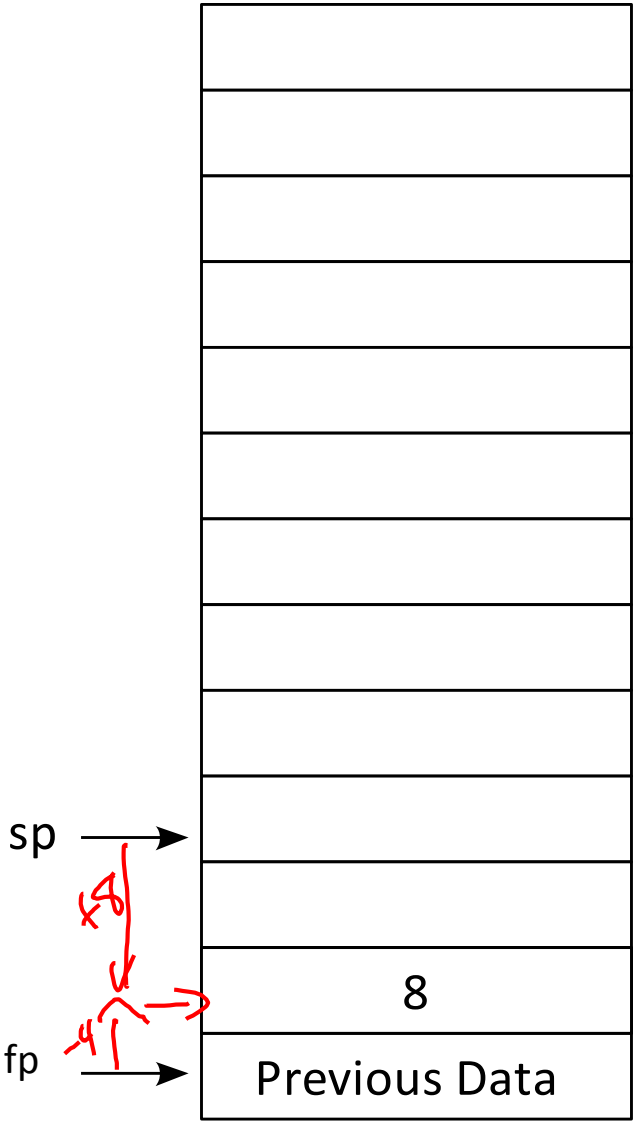
two inputs
one output

```
20   SUB    sp, sp #12
```

SUB sp, sp, #8
STR
STR

sub sp, sp, #4
STR
SUB
STR

Multiply_by_Adding Stack Contents



fp

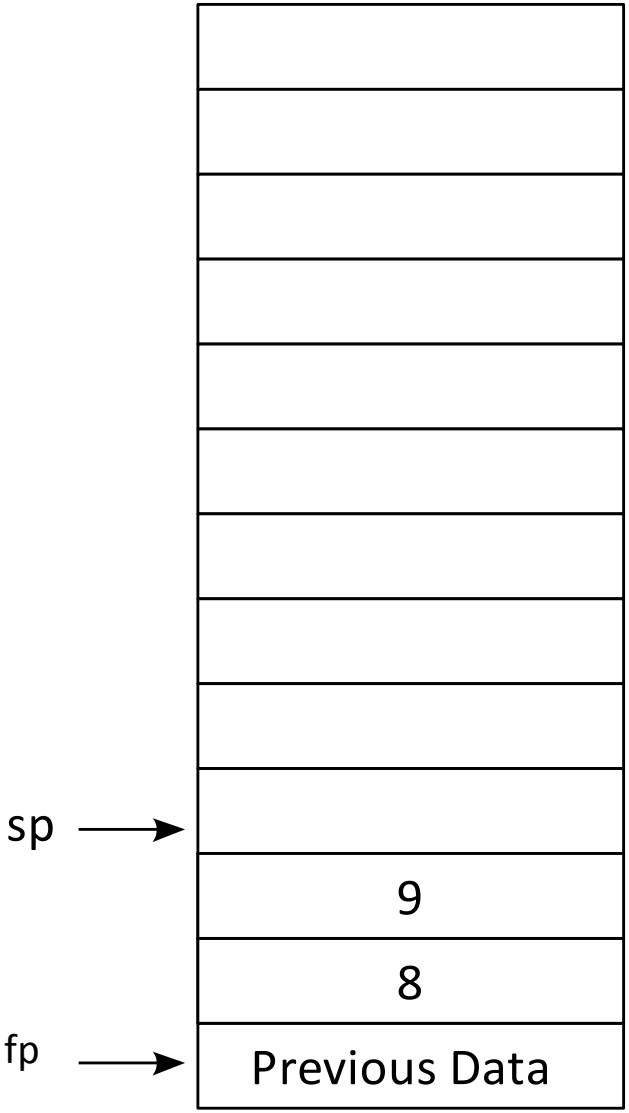
```
24  STR r1, [fp, #-4]
```

sp

STR r1, [sp, #8]

Push num1 to the stack

Multiply_by_Adding Stack Contents



fp

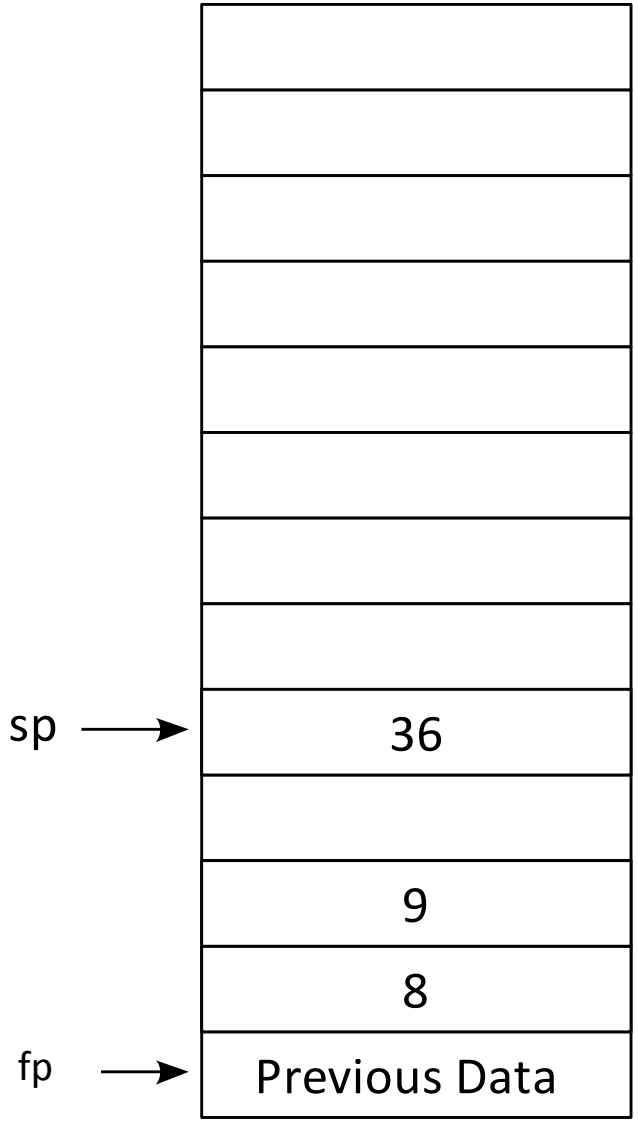
```
28  STR r2, [fp, #-8]
```

sp

STR r2, [sp, #4]

Push num2 to the stack

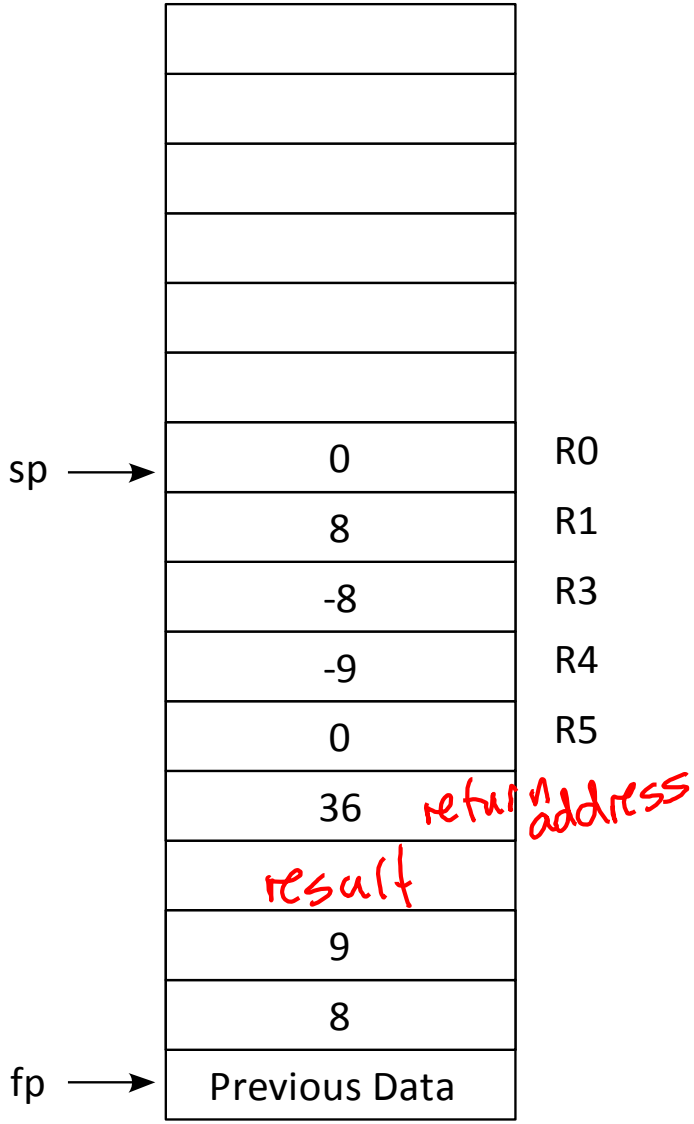
Multiply_by_Adding Stack Contents




```
52   PUSH   {lr}
```

store link register value before branching and linking

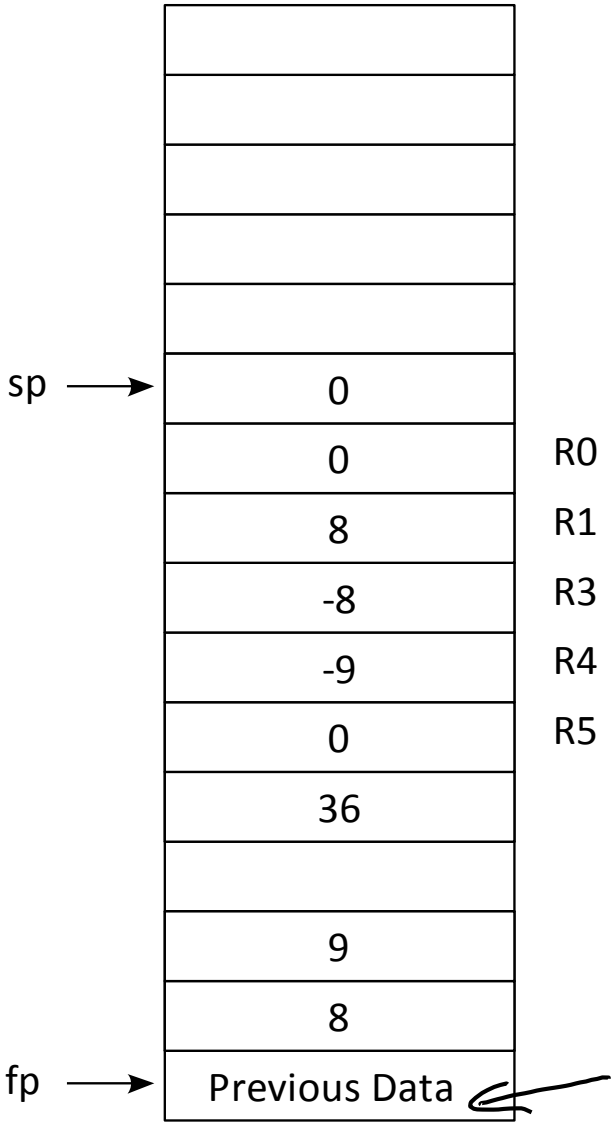
Multiply_by_Adding Stack Contents




```
56  PUSH  {r0, r1, r3, r4, r5}
```

store old register contents before putting new values in registers

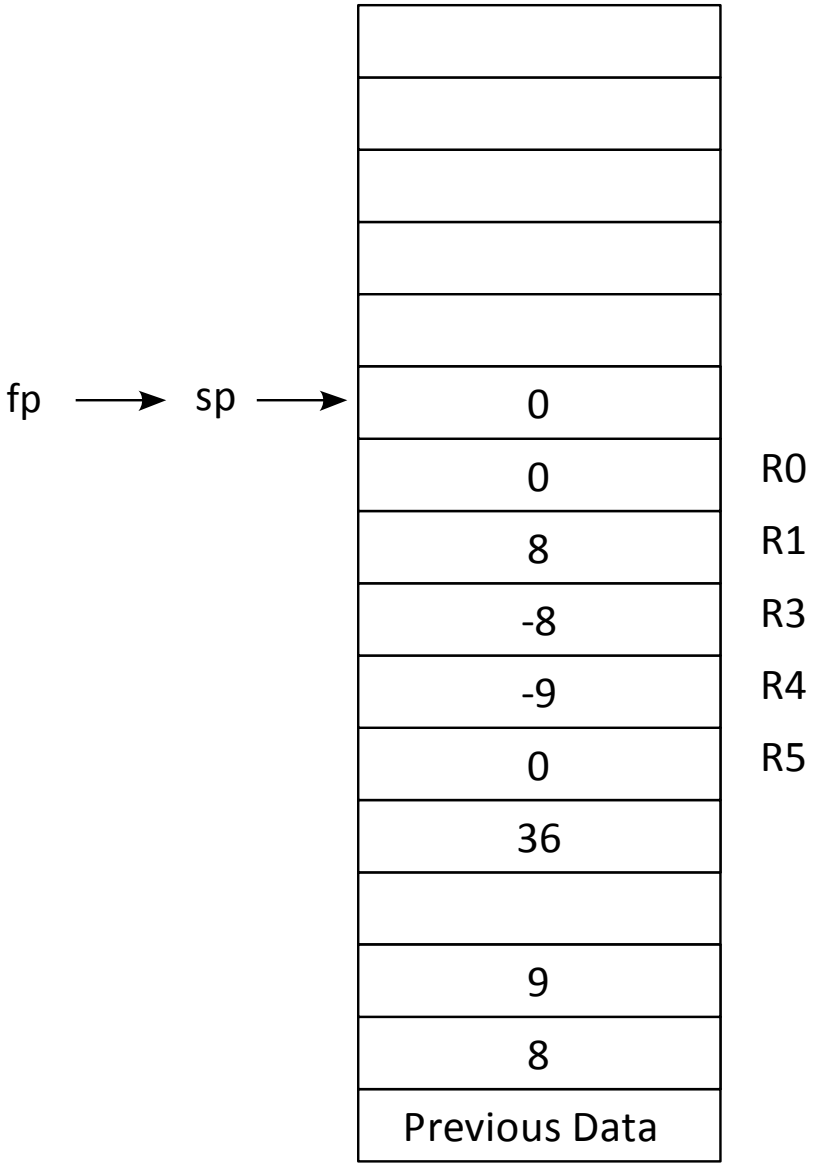
Multiply_by_Adding Stack Contents




```
88   PUSH   {fp}
```

store fp of multiply by adding routine before going to abs

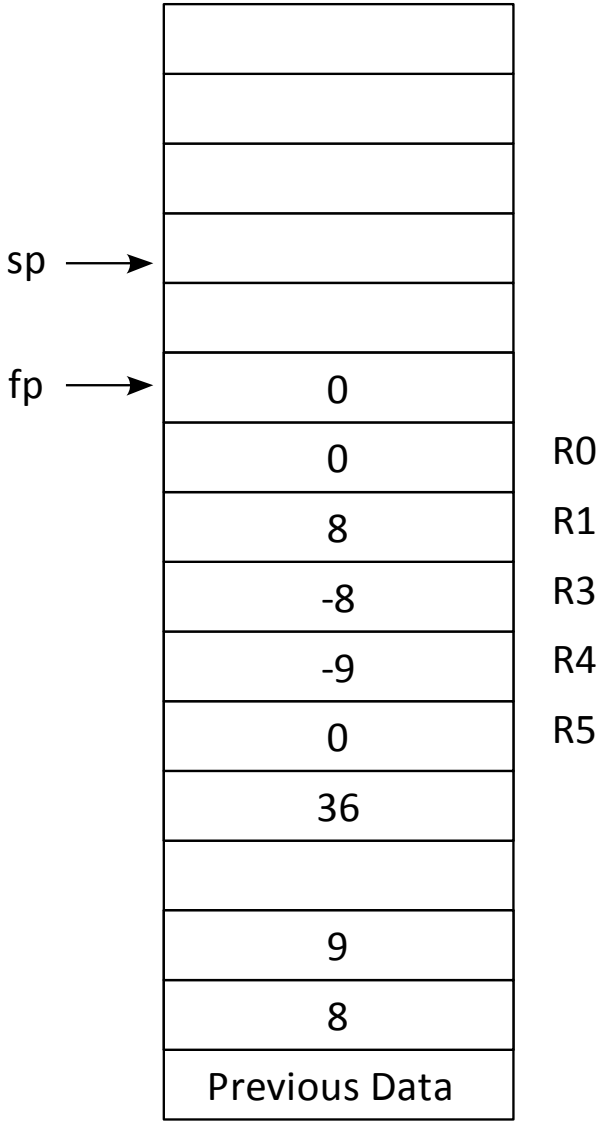
Multiply_by_Adding Stack Contents




```
92   MOV   fp, sp
```

Establish fp for abs routine

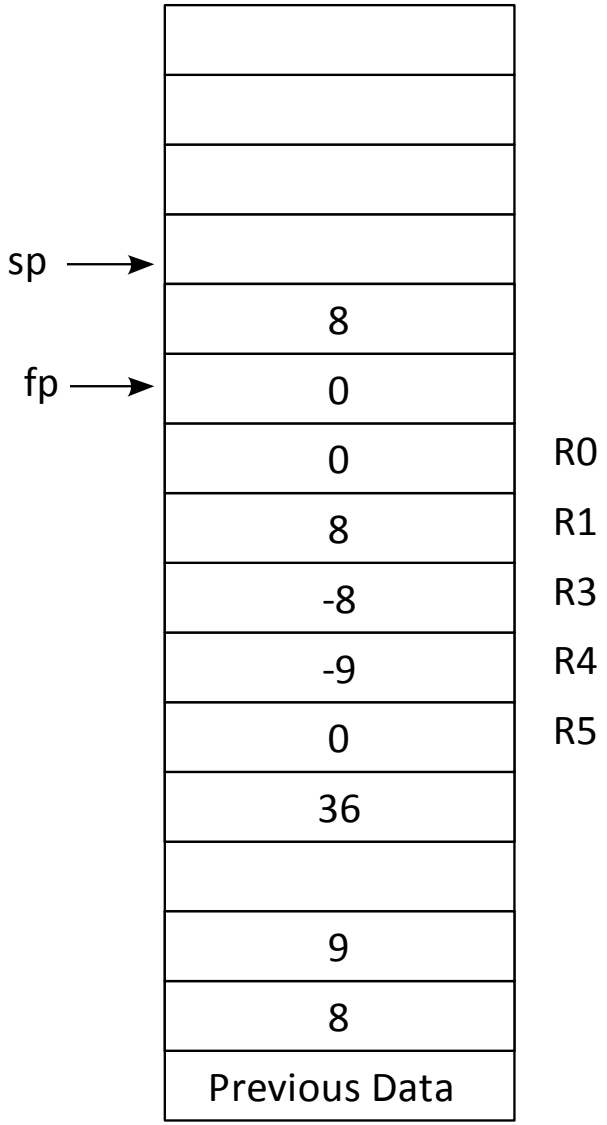
Multiply_by_Adding Stack Contents




```
96  SUB  sp, sp #8
```

abs routine
one input
one output,
make room on
stack

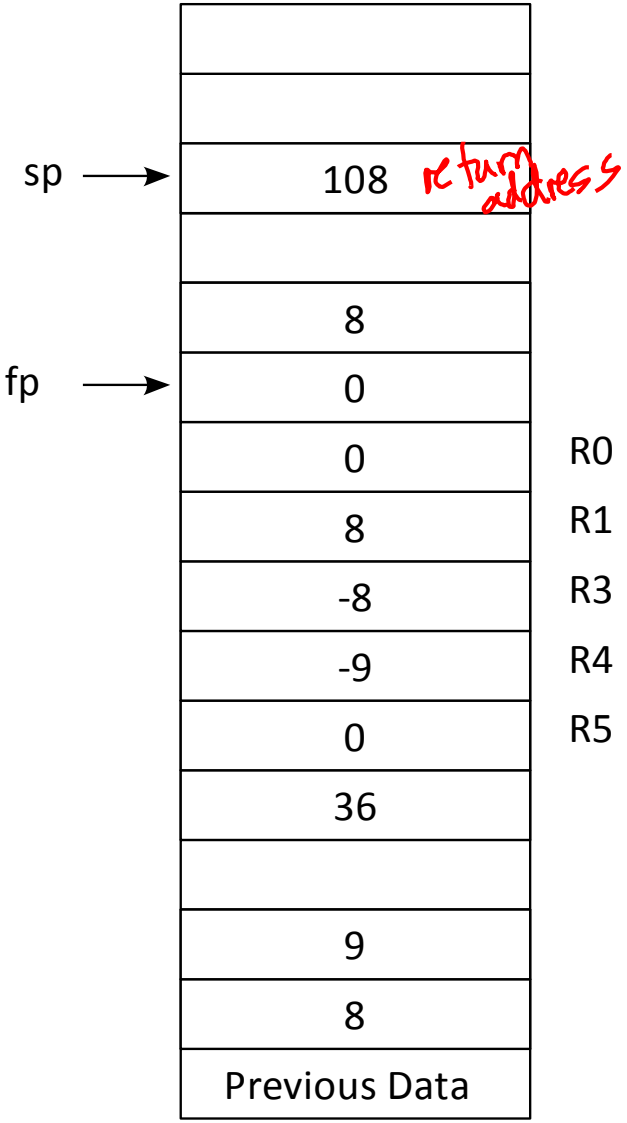
Multiply_by_Adding Stack Contents




```
100 STR r1, [fp, #-4]
```

Store argument for abs

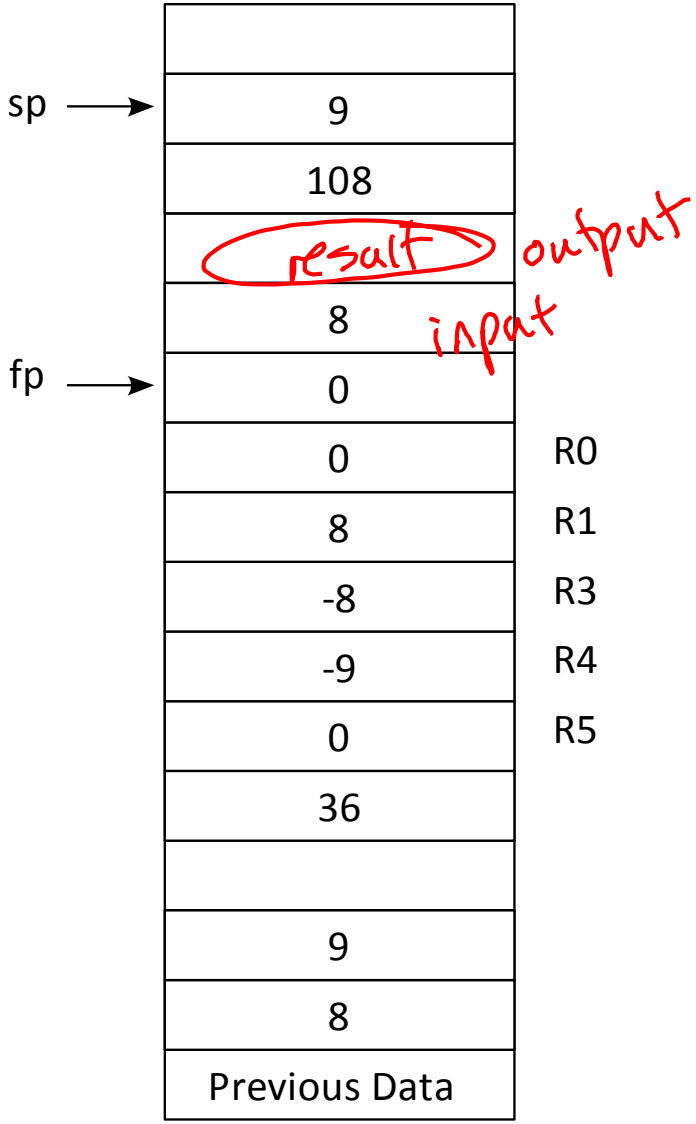
Multiply_by_Adding Stack Contents




```
196   PUSH {lr}
```

push link register before branching and linking

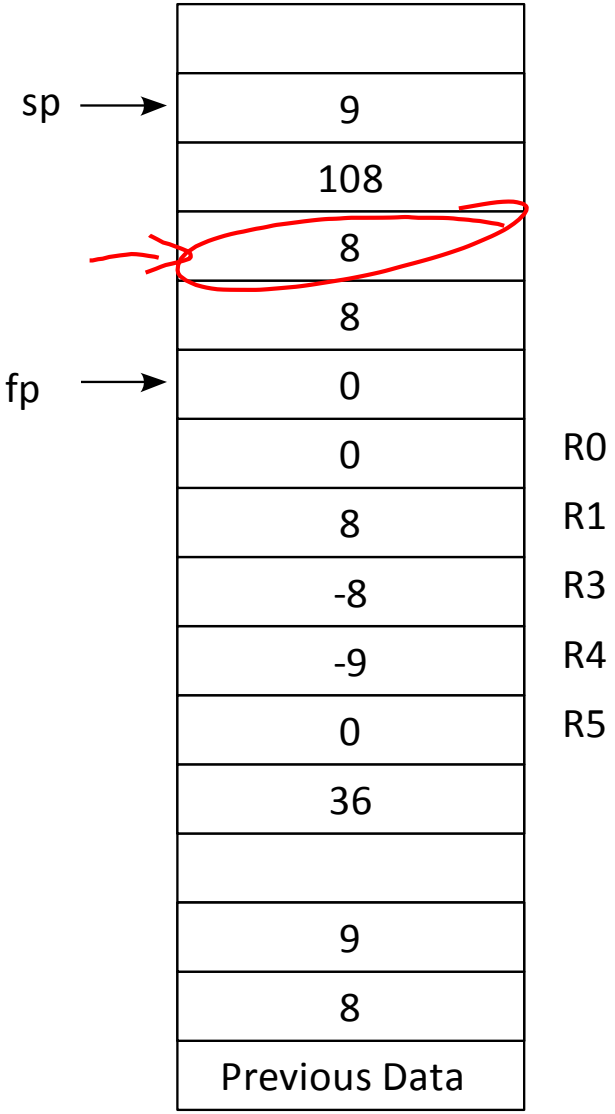
Multiply_by_Adding Stack Contents




```
200   PUSH   {r0}
```

save old value of r0 so I can use r0

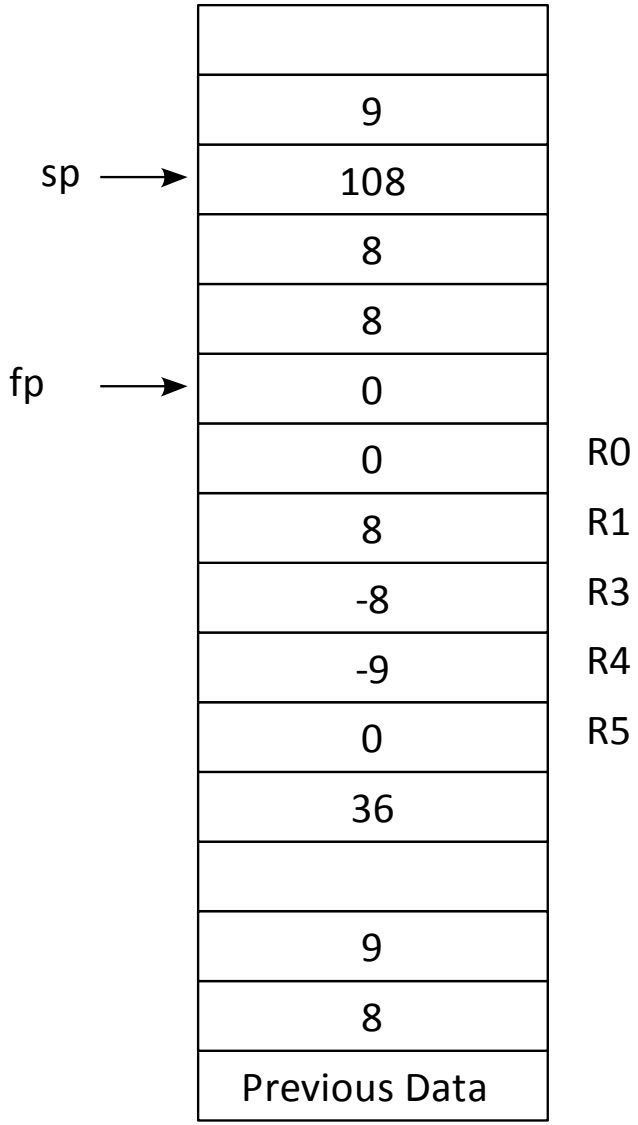
Multiply_by_Adding Stack Contents




```
220   STR   r0, [fp, #-8]
```

store return value from abs

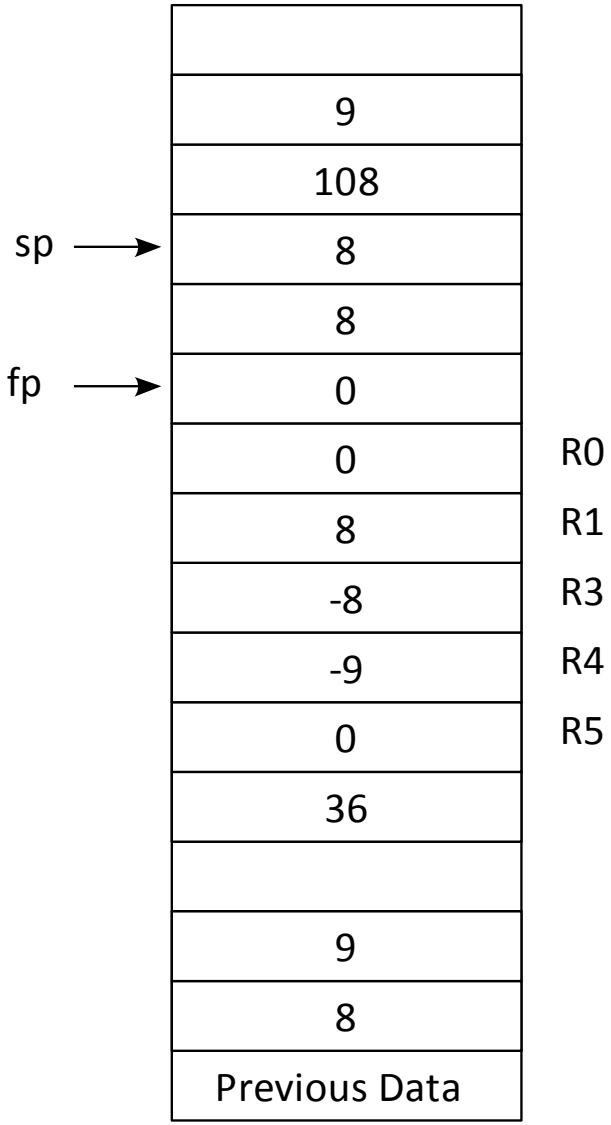
Multiply_by_Adding Stack Contents




```
224   POP   {r0}
```

restore pre-abs value of r0

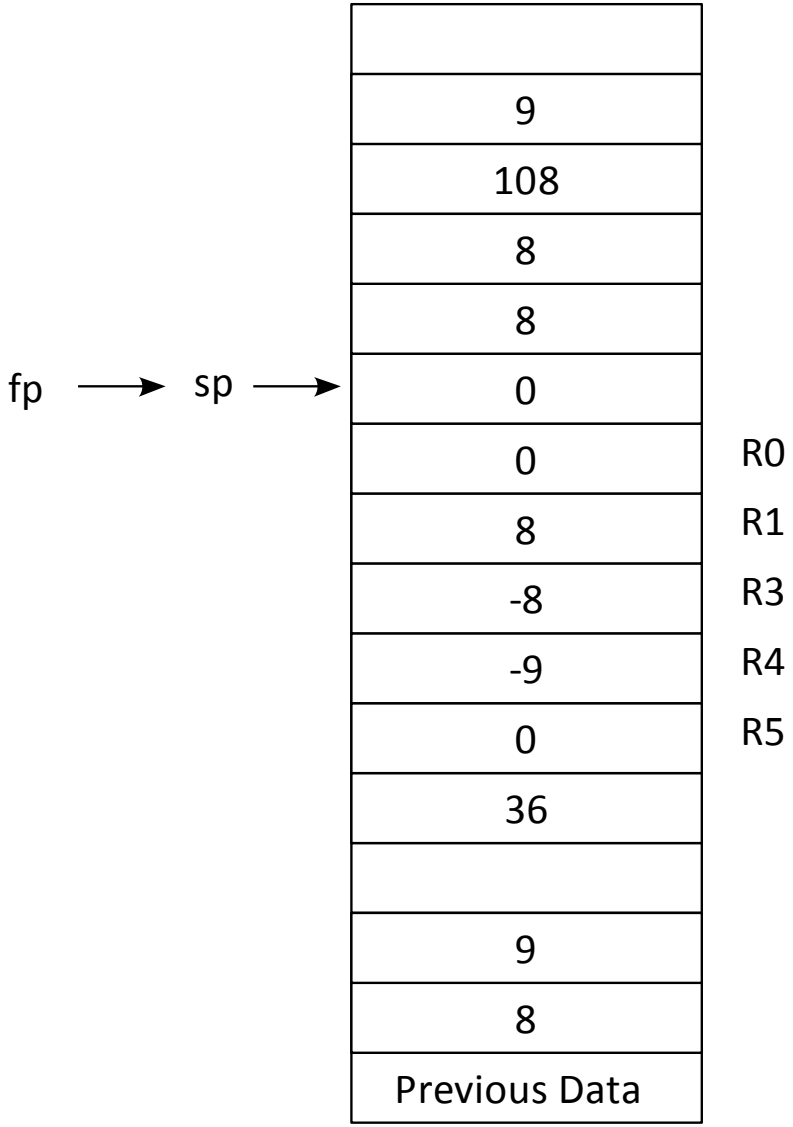
Multiply_by_Adding Stack Contents




```
228   POP    {pc}
```

takes me to the instruction at address 108

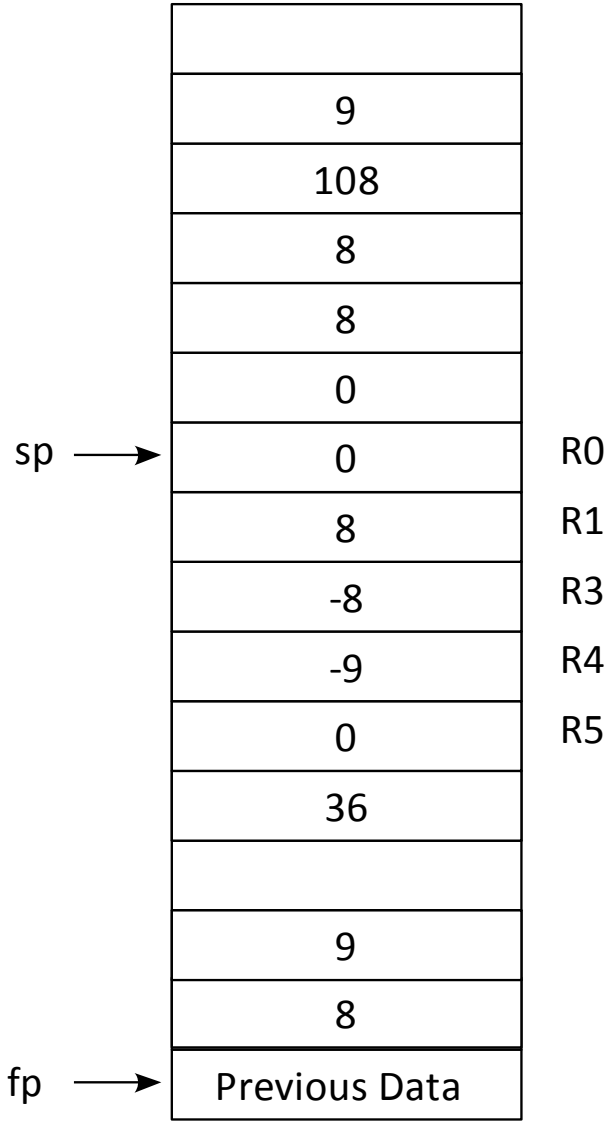
Multiply_by_Adding Stack Contents




```
112   MOV   sp, fp
```

release space allocated for abs

Multiply_by_Adding Stack Contents

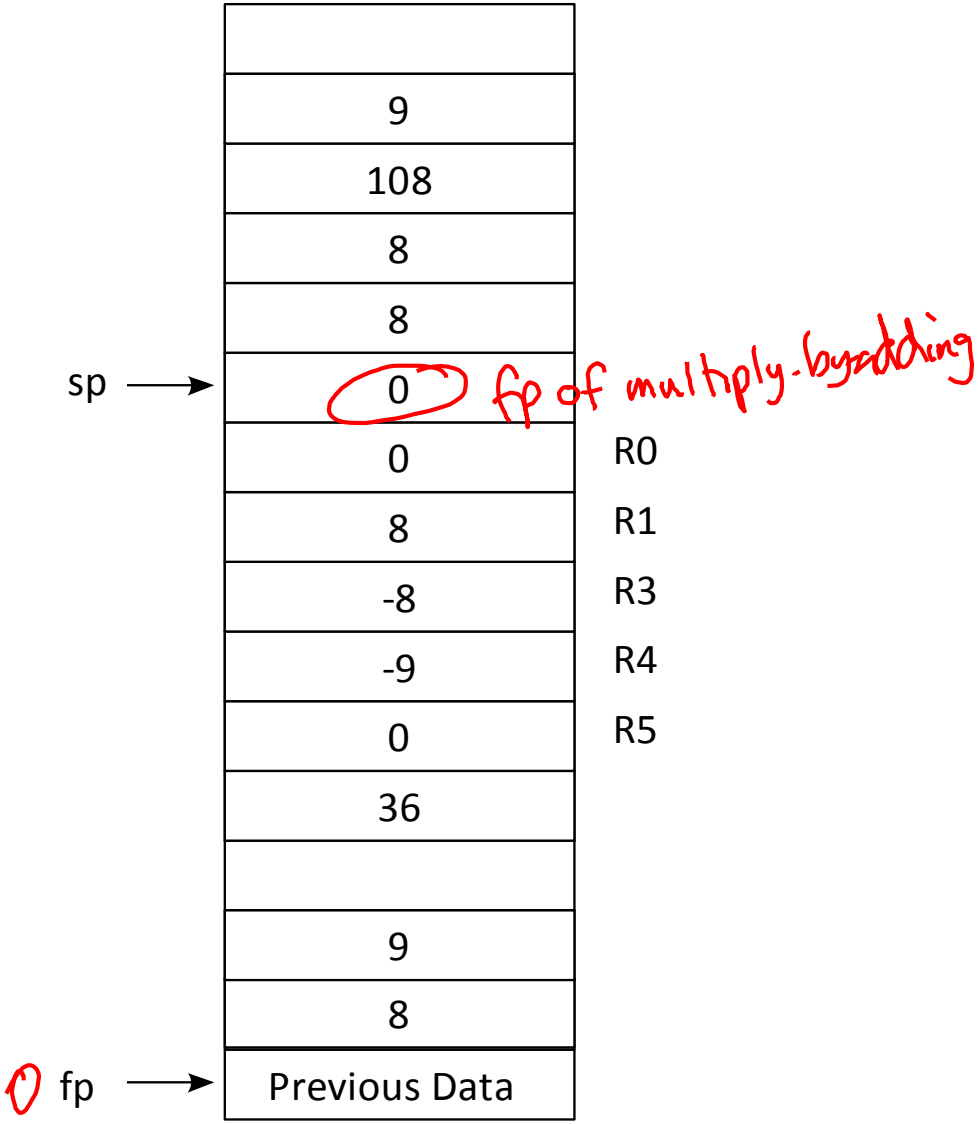



```
116   POP   {fp}
```

go back to previous frame pointer

Multiply-by-adding

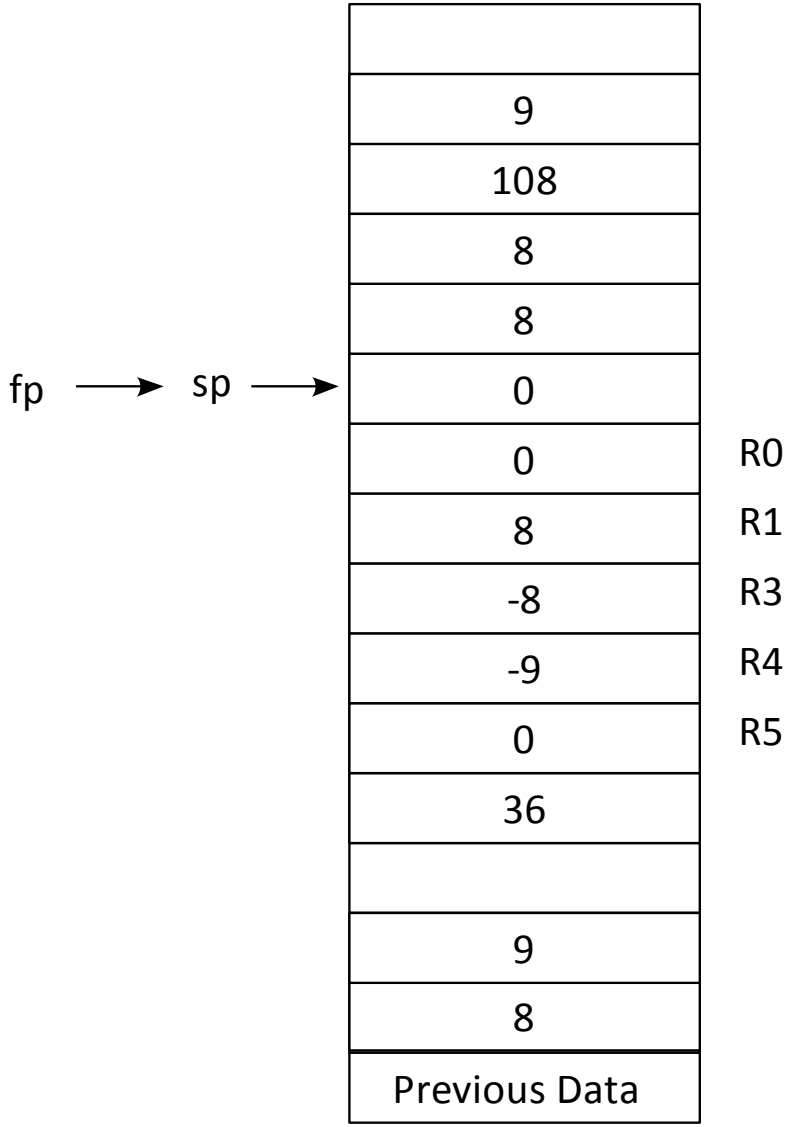
Multiply_by_Adding Stack Contents




```
120   PUSH   {fp}
```

before calling abs for the second time, store fp of multiply by adding

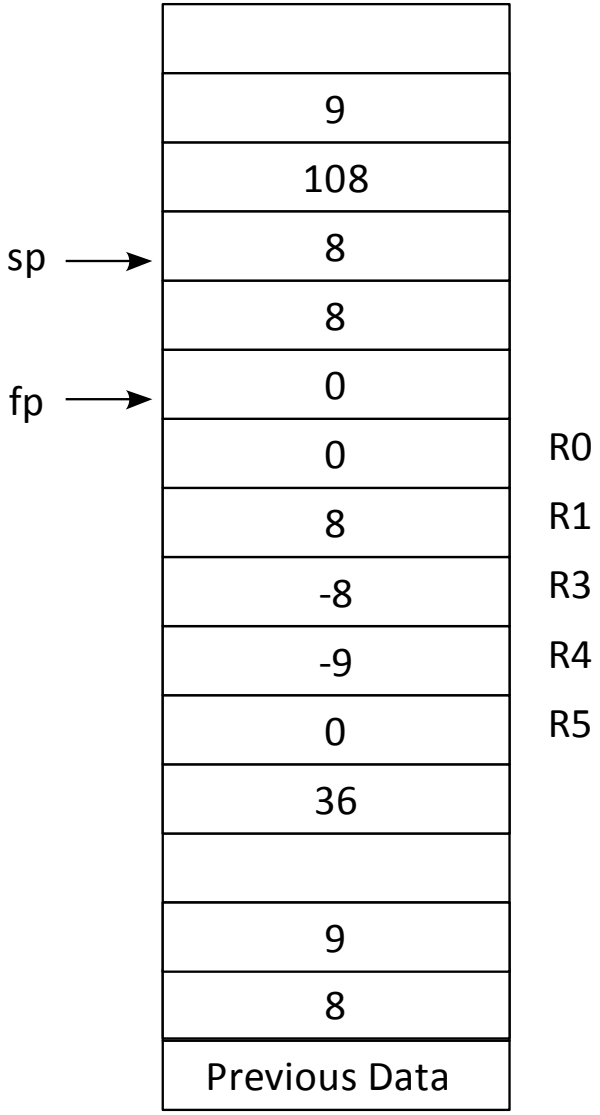
Multiply_by_Adding Stack Contents




```
124   MOV   fp, sp
```

Having stored the multiply by adding fp, establish abs fp

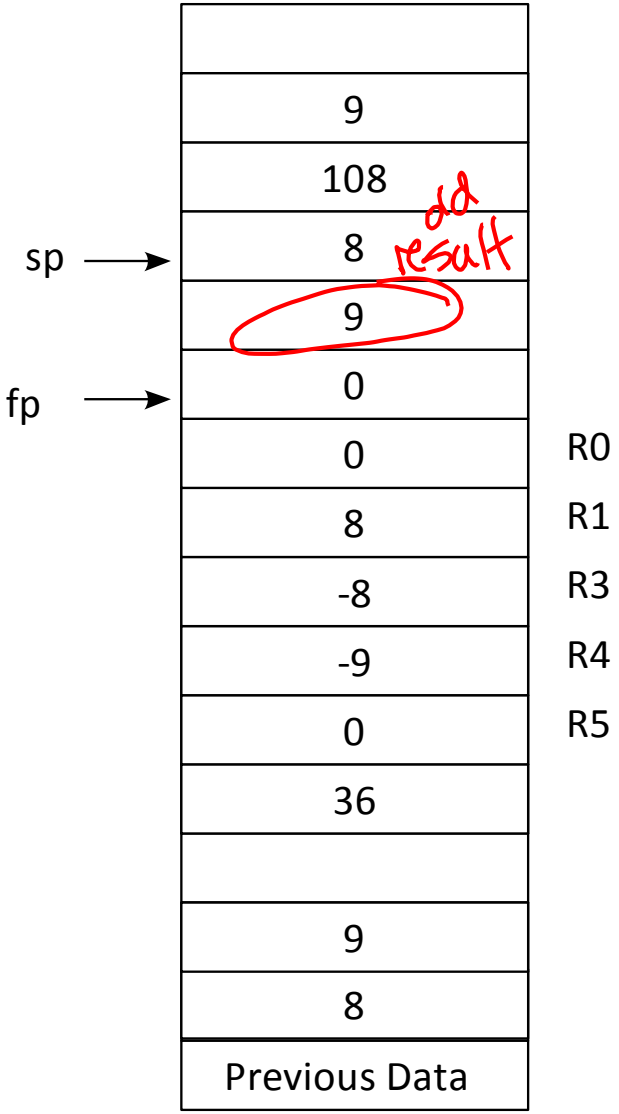
Multiply_by_Adding Stack Contents




```
128  SUB  sp, sp #8
```

make room for input and output

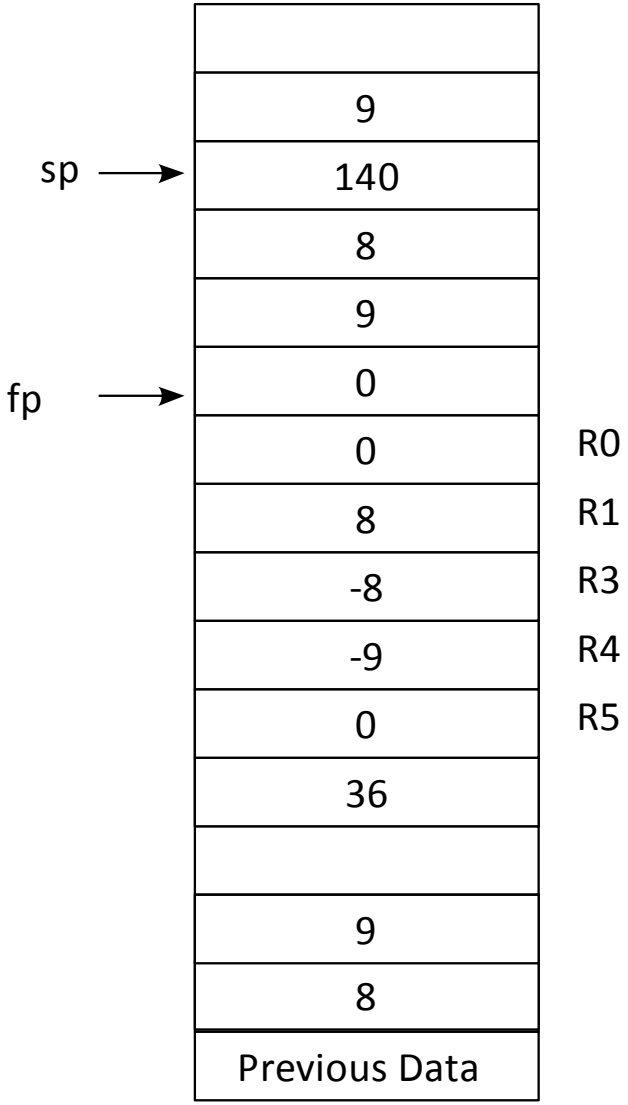
Multiply_by_Adding Stack Contents




```
132   STR r0, [fp, #-4]
```

place argument in allocated space

Multiply_by_Adding Stack Contents




```
196   PUSH   {lr}
```

save return address

Multiply_by_Adding Stack Contents

activation frame of abs

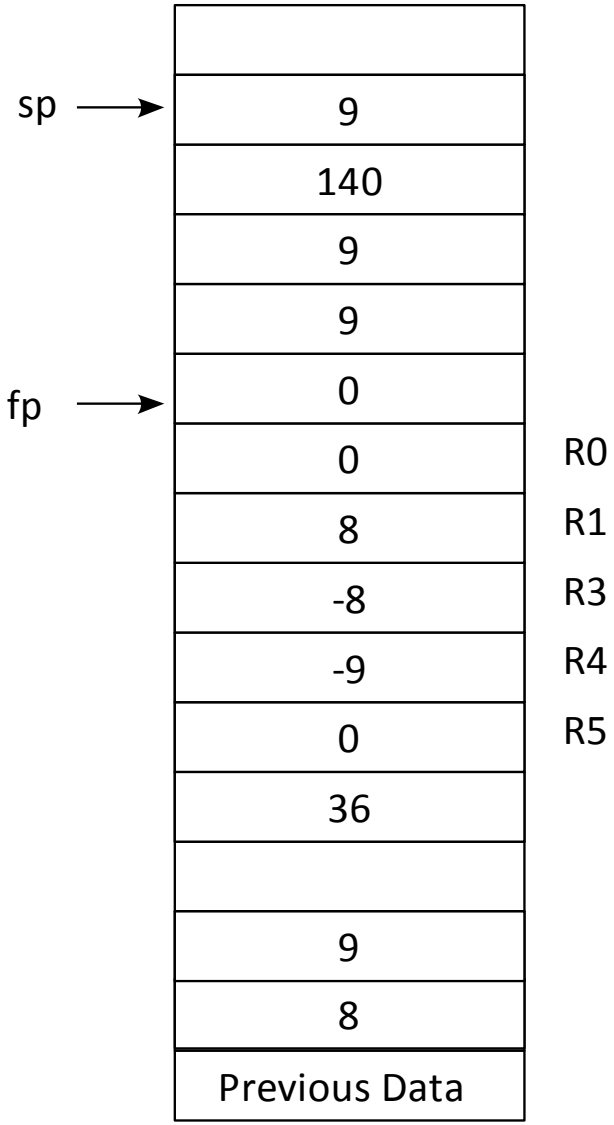
activation frame of multiply by adding

sp →	9	
	140	
	8	
	9	
fp →	0	
	0	R0
	8	R1
	-8	R3
	-9	R4
	0	R5
	36	
	9	
	8	
	Previous Data	

```
200   PUSH   {r0}
```

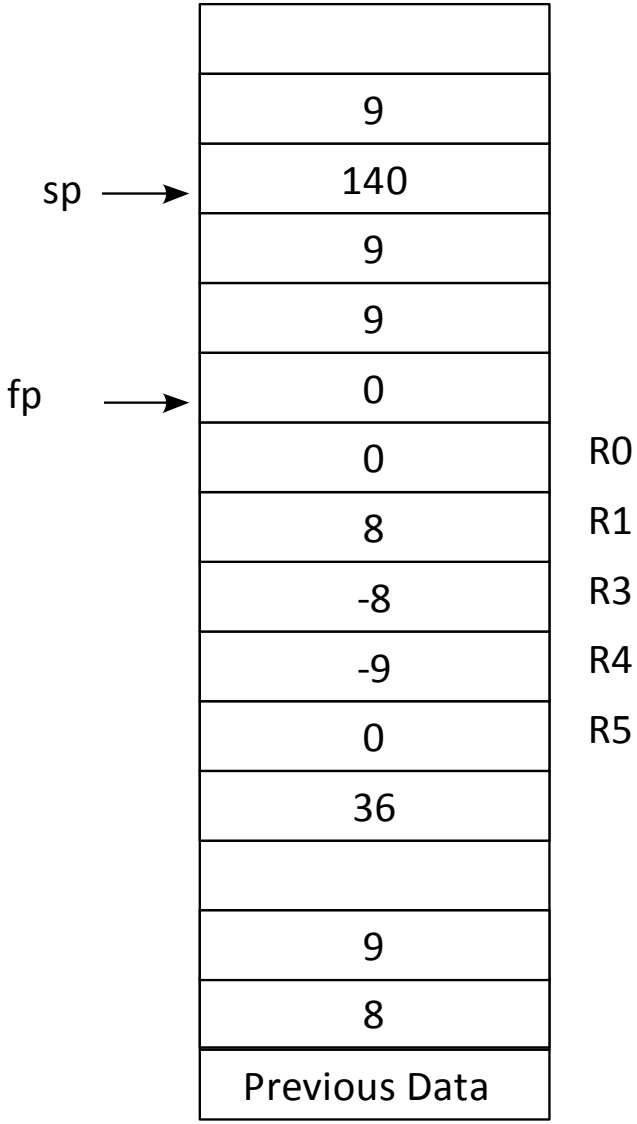
save old value of r0 in stack because r0 can be used by abs

Multiply_by_Adding Stack Contents




```
220  STR r0, [fp, #-8]
```

Multiply_by_Adding Stack Contents

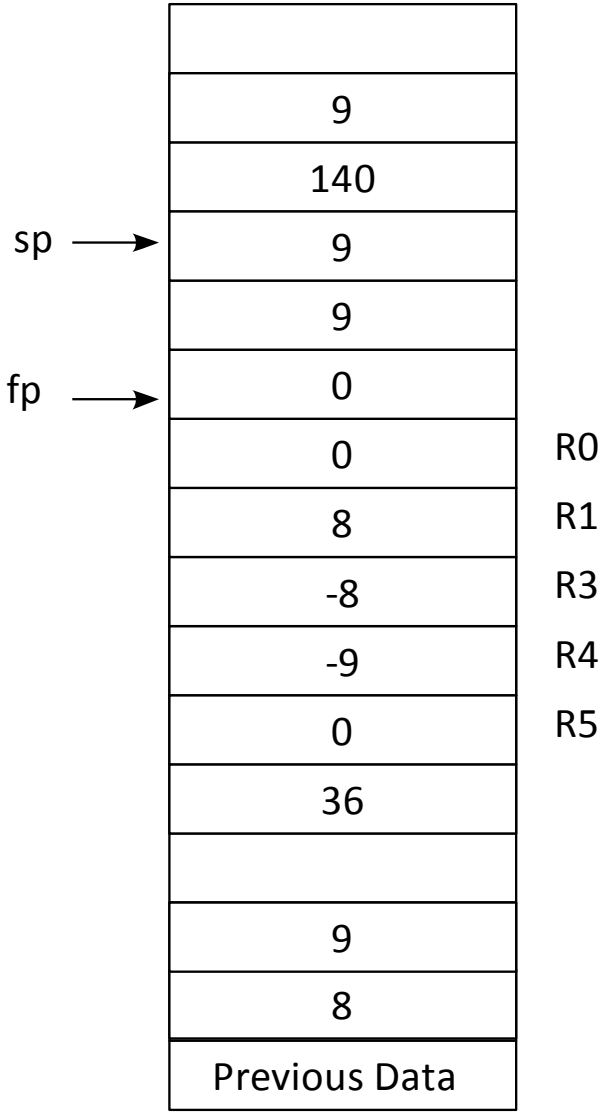



```
224   POP   {r0}
```

restore old value of r0

multiply by adding

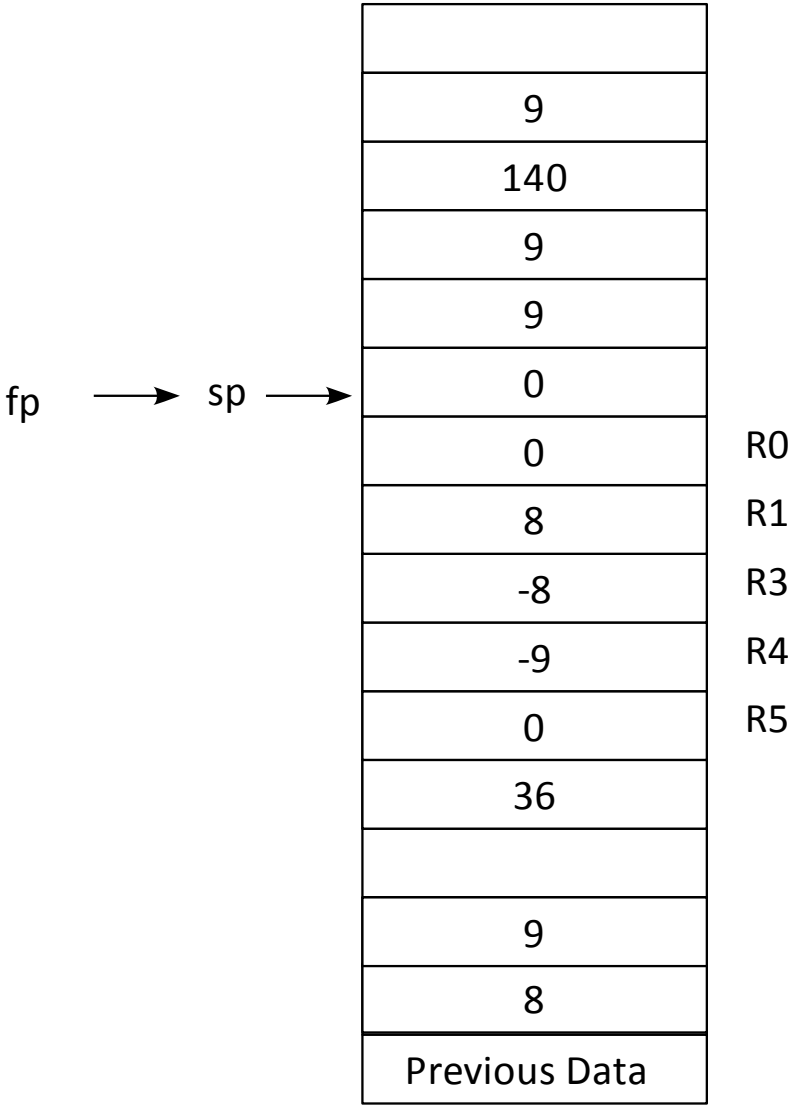
Multiply_by_Adding Stack Contents




```
228   POP   {pc}
```

return from abs

Multiply_by_Adding Stack Contents

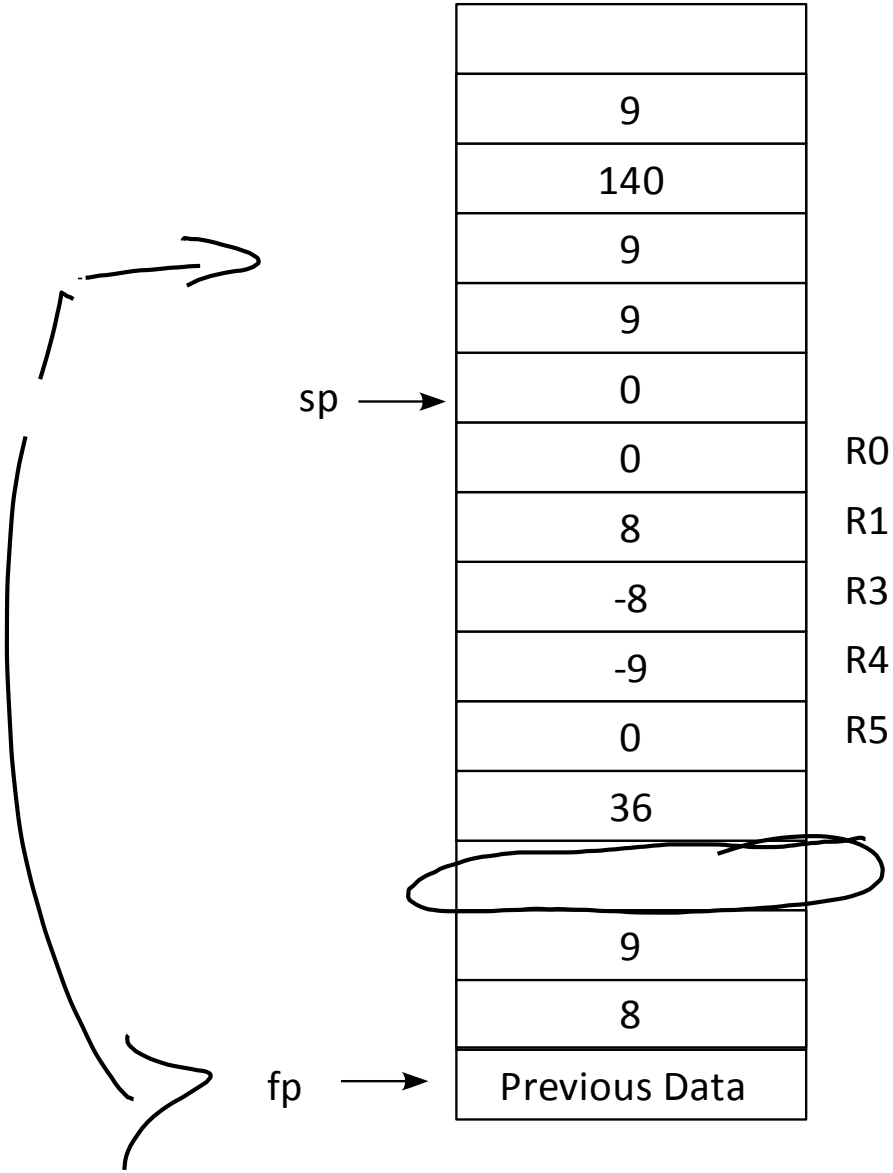



```
144  MOV   sp, fp
```

collapse activation frame

release allocated space

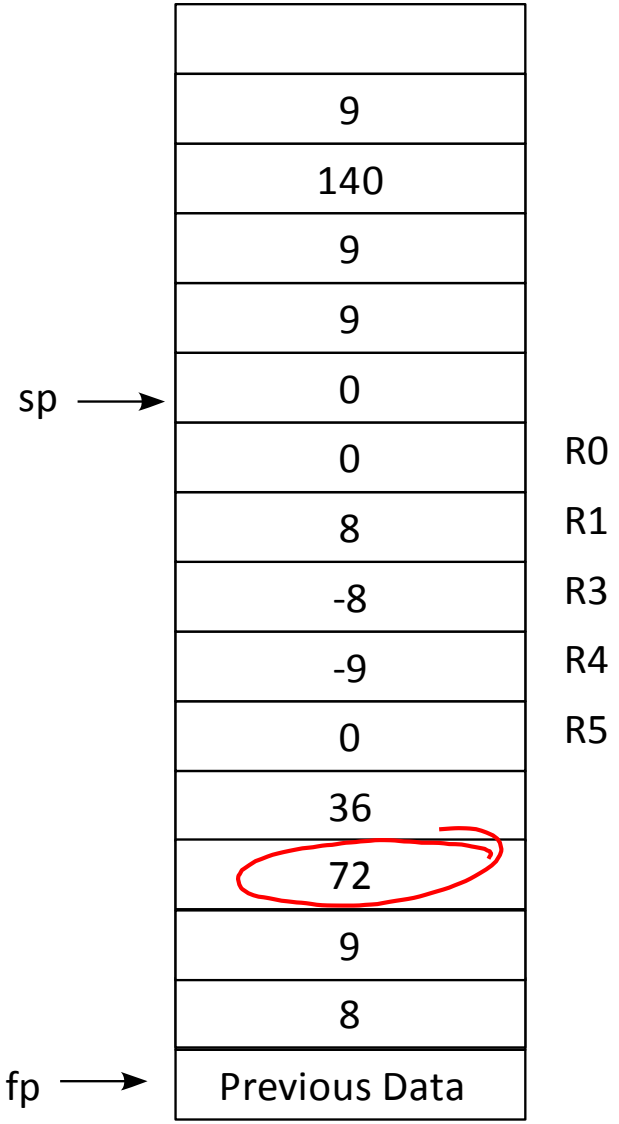
Multiply_by_Adding Stack Contents




```
148   POP   {fp}
```

adjust frame pointer to multiply by adding fp

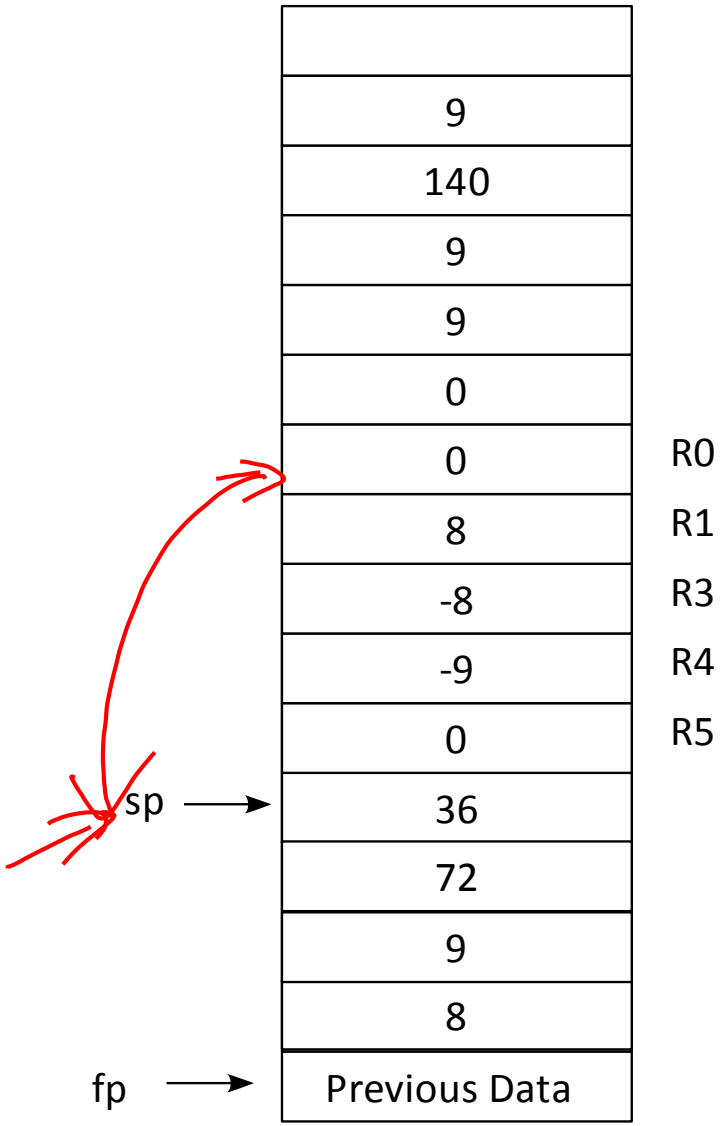
Multiply_by_Adding Stack Contents




```
184  STR r3, [fp, #-12]
```

store the result

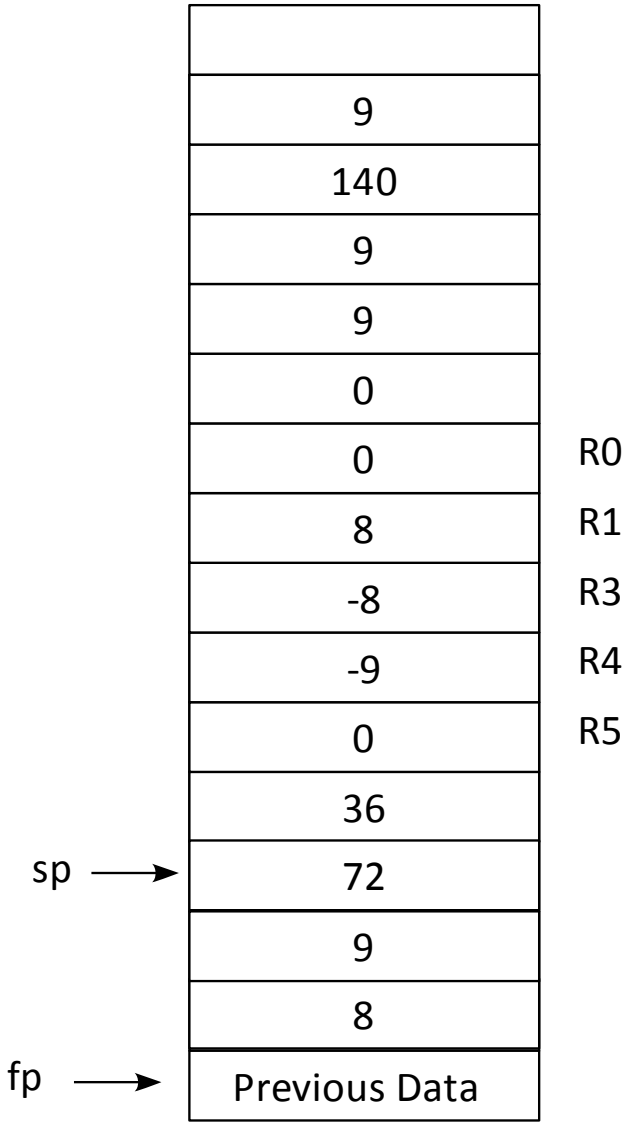
Multiply_by_Adding Stack Contents




```
188   POP   {r0, r1, r3, r4, r5}
```

restore the previous values of r0, r1, r3, r4, r5

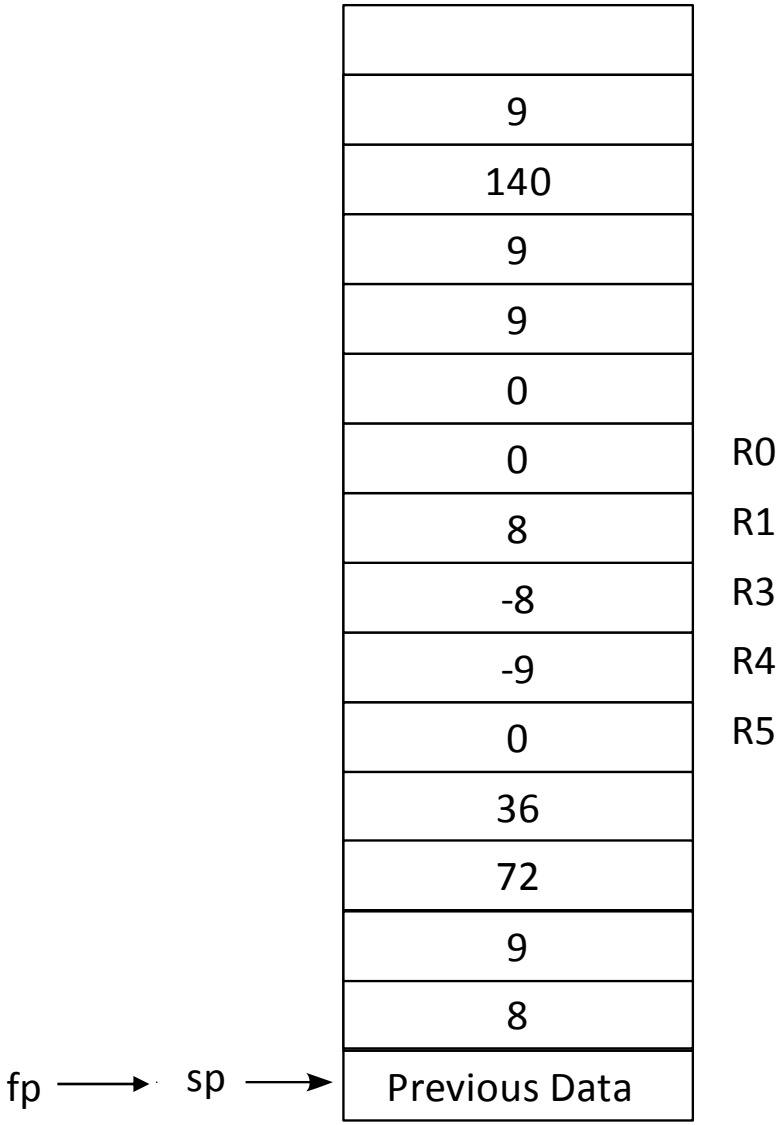
Multiply_by_Adding Stack Contents




```
192   POP   {pc}
```

return from multiply by adding

Multiply_by_Adding Stack Contents




```
44   MOV   sp, fp
```

releasing allocated space

4.1 Swap – Pass by Reference, C++

```
void swap (int*, int*);

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

void swap (int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

&x address

*a data

4.1 Swap – Call by Reference, ASM

```
        MOV     sp, #0x00000000  ; Initialize the stack pointer(0)
        MOV     fp, #0xFFFFFF00  ; Initialize the frame pointer (4)
        B       main             ; Go to main(8)
swap    PUSH    {fp}             ; Store old frame pointer value(12)
        MOV     fp, sp           ; Set frame pointer for swap(16)
        SUB     sp, sp, #4       ; Allocate space for temp (20)
        LDR     r1, [fp, #4]     ; Get address of parameter a (24)
        LDR     r2, [r1]         ; Get value of parameter a (28)
        STR     r2, [fp, #-4]    ; Store a in temp in stack frame (32)
        LDR     r0, [fp, #8]     ; Get address of parameter b (36)
        LDR     r3, [r0]         ; Get value of parameter b (40)
        STR     r3, [r1]         ; store parameter b in parameter a (44)
        LDR     r3, [fp, #-4]    ; Get temp from stack frame (48)
        STR     r3, [r0]         ; Store temp in b (52)
        MOV     sp, fp           ; Collapse stack frame (56)
        POP     {fp}             ; Restore previous frame pointer (60)
        MOV     pc, lr           ; Return from swap (64)
main    PUSH    {fp}             ; Store old frame pointer value (68)
        MOV     fp, sp           ; Set frame pointer for main (72)
        SUB     sp, sp, #8       ; Allocate space on stack (76)
        ADR     r6, x            ; Put &x in r6 (80)
        STR     r6, [fp, #-4]    ; Store &x on the stack (84)
        ADR     r6, y            ; Put &y in r6 (88)
        STR     r6, [fp, #-8]    ; store &y on the stack (92)
        BL      swap             ; Go to stack (96)
        MOV     sp, fp           ; Collapse frame (100)
        POP     {fp}             ; Restore old frame pointer (104)
Stop    B       Stop
x       DCD     2
y       DCD     3
```

Swap by Reference Stack, Registers, Memory

```
0    MOV    sp, #0x00000000
```

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC		
FFFF FFF0		
FFFF FFF4		
FFFF FFF8		
FFFF FFFC		

sp →

Swap by Reference Stack, Registers, Memory

{fp}

PUSH 2 steps

sp ← sp - 4

M[sp] ← fp

```
64   PUSH    {fp}
```

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC		
FFFF FFF0		
FFFF FFF4		
FFFF FFF8		
FFFF FFFC	original fp	0xFFFF FF00

sp →

Swap by Reference Stack, Registers, Memory

fp = 0xFFFFFFFC

```
68   MOV    fp, sp
```

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC		
FFFF FFF0		
FFFF FFF4		
FFFF FFF8		
fp->sp-> FFFF FFFC	original fp	0xFFFF FF00

Swap by Reference Stack, Registers, Memory

```
72   SUB    sp, sp, #8
```

Make room for two input parameters

	Address	Meaning	Value
	FFFF FFE8		
	FFFF FFEC		
	FFFF FFF0		
sp →	FFFF FFF4		
	FFFF FFF8		
main fp →	FFFF FFFC	original fp	0xFFFF FF00

Swap by Reference Stack, Registers, Memory

Write first input parameter to stack

```
80   STR    r6, [fp, #-4]
```

	Address	Meaning	Value
	FFFF FFE8		
	FFFF FFEC		
	FFFF FFF0		
sp →	FFFF FFF4		
	FFFF FFF8	&x	112
fp →	FFFF FFFC	original fp	0xFFFF FF00

Swap by Reference Stack, Registers, Memory

```
88   STR    r6[fp, #-8]
```

Write second input parameter to stack

	Address	Meaning	Value
	FFFF FFE8		
	FFFF FFEC		
	FFFF FFF0		
sp →	FFFF FFF4	&y	116
	FFFF FFF8	&x	112
fp →	FFFF FFFC	original fp	0xFFFF FF00

Swap by Reference Stack, Registers, Memory

```
8    PUSH    {fp}
```

	Address	Meaning	Value	
	FFFF FFE8			
	FFFF FFEC			
sp →	FFFF FFF0	main fp	0xFFFF FF FC	main fp
	FFFF FFF4	&y	116	
	FFFF FFF8	&x	112	
fp →	FFFF FFFC	original fp	0xFFFF FF00	

Swap by Reference Stack, Registers, Memory

```
12   MOV    fp, sp
```

When going to a new routine you may store old frame pointer and return address on stack.

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC		
FFFF FFF0	main fp	0xFFFF FFFC
FFFF FFF4	&y	116
FFFF FFF8	&x	112
FFFF FFFC	original fp	0xFFFF FF~~00~~

sp → swap fp →

You may also store the return address in the link register instead

Swap by Reference Stack, Registers, Memory

making room for temp

```
16  SUB    sp, sp, #4
```

	Address	Meaning	Value
	FFFF FFE8		
sp →	FFFF FFEC		
swap fp →	FFFF FFF0	main fp	0xFFFF FFFC
	FFFF FFF4	&y	116
	FFFF FFF8	&x	112
	FFFF FFFC	original fp	0xFFFF FFØØ

Swap by Reference Stack, Registers, Memory

```
28  STR    r2, [fp, #-4]
```

	Address	Meaning	Value
	FFFF FFE8		
sp →	FFFF FFEC	temp place for y	3
fp →	FFFF FFF0	main fp	0xFFFF FFFC
	FFFF FFF4	&y	116
	FFFF FFF8	&x	112
	FFFF FFFC	original fp	0xFFFF FFF0

Swap by Reference Stack, Registers, Memory

Collapse stack frame

```
52   MOV    sp, fp
```

sp => fp →

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC	temp	3
FFFF FFF0	main fp	0xFFFF FFFC
FFFF FFF4	&y	116
FFFF FFF8	&x	112
FFFF FFFC	original fp	0xFFFF FFC0

Swap by Reference Stack, Registers, Memory

POP fp
fp ← M[sp]
sp ← sp + 4

```
56   POP    {fp}
```

fp ← 0xFFFF FFFC

	Address	Meaning	Value
	FFFF FFE8		
	FFFF FFEC	temp	3
	FFFF FFF0	main fp	0xFFFF FFFC
sp →	FFFF FFF4	&y	116
	FFFF FFF8	&x	112
fp →	FFFF FFFC	original fp	0xFFFF FF00

Swap by Reference Stack, Registers, Memory

```
96   MOV    sp, fp
```

release stack frame allocated to main

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC	temp	3
FFFF FFF0	main fp	0xFFFF FFFC
FFFF FFF4	&y	116
FFFF FFF8	&x	112
FFFF FFFC	original fp	0xFFFF FF00

sp -> fp ->

Swap by Reference Stack, Registers, Memory

i=0;
a=b/i;

```
100 POP    {fp}
```

Restore original fp value

fp = 0xFFFF FFØØ

sp = 0x0000 0000

Address	Meaning	Value
FFFF FFE8		
FFFF FFEC	temp	3
FFFF FFF0	main	0xFFFF FFFC
FFFF FFF4	&y	116
FFFF FFF8	&x	112
FFFF FFFC	original fp	0xFFFF FFDD

sp →

4.2 Exception Overview

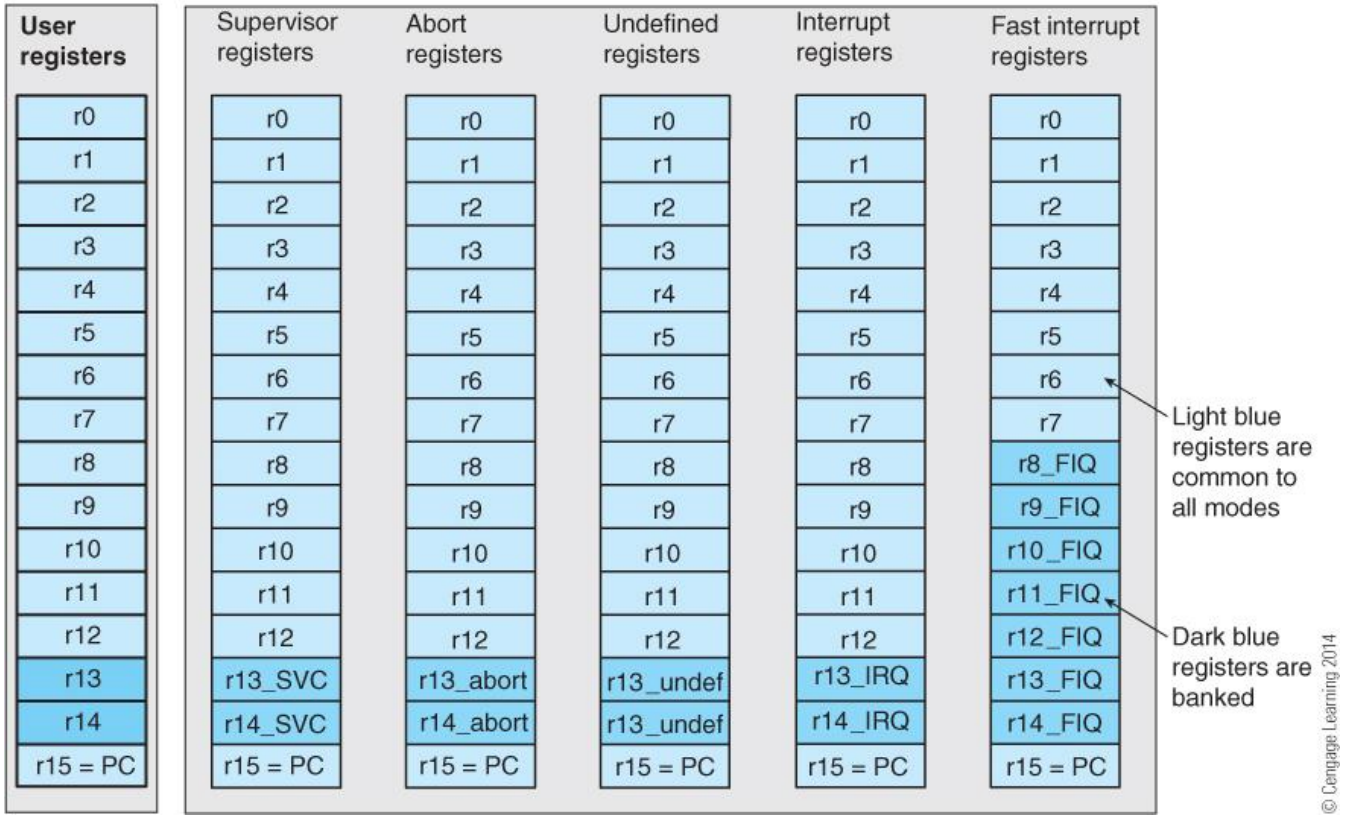
- Exceptions are like subroutines that are jammed into code at runtime.
- Exceptions use call and return mechanisms similar to subroutines; the major difference being that the call address is supplied by the hardware.
- Typically, a processor decodes the exception type and reads a pointer that indicates the start of the exception handling routine. Some processors save the current status word plus 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.

4.2 Privileged Modes and Exceptions

- Exceptions are events that force the computer to stop normal processing and to invoke a program called an __exception__ __handler__(operating system).
- An ARM processor has several operating modes described below.
- The __five__ __lower__-__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.
- Different modes have different stack pointer and link registers.

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

4.2 Switching Modes Fast – Register Banks



- Registers in dark blue are banked and associated with specific 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.

4.2 ARM Exception Handling

- 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 ~~s~~s 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.

user → operating
ra → lr
CPSR → SPSR

operating ← user
pc ← lr
CPSR ← SPSR

4.2 ARM Exception Vectors

- Interrupt requests are __disabled__ by setting bit __7__ of the __CPSR__. If the current exception is a fast ___ ~~6 of the~~ _____. interrupt request, further FIQ interrupts 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 Exception Vectors

Exception	Mode	Vector Address
Reset	SVC	0x00000000 — Branch to reset handler
Undefined instruction	UND	0x00000004 — branch to undefined instruction handle
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 —

4.2 ARM Return from Exception

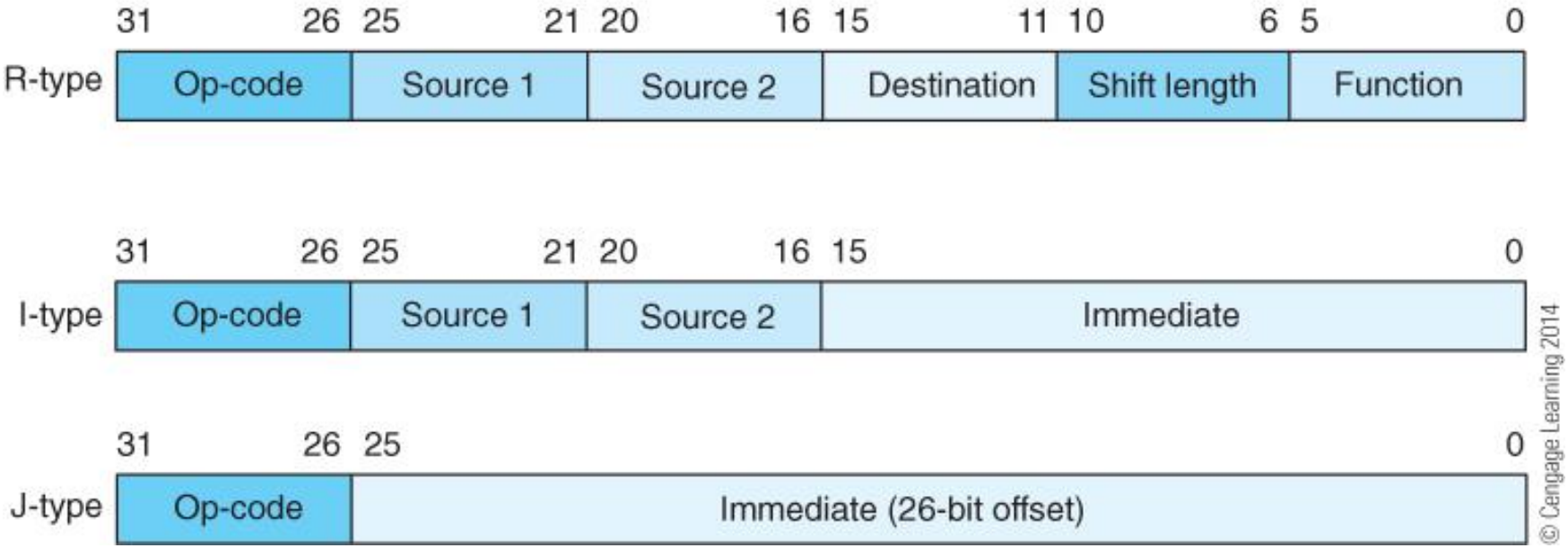
- After the exception has been dealt with by a suitable handler, it is necessary to __return__ to the point at which the exception was called (of course, if the exception was __fatal__ a __return__ is no longer possible).
- In order to __return__ from an exception, the information that defines the pre-exception 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__ to restore the __CPSR__.

You need both to happen at the same time so we have special instructions that take care of this.

4.3 MIPS: Another RISC

- MIPS is a RISC architecture developed by John Hennessy at _____ University in 1980 to exploit the best aspects of RISC philosophy in an efficient ___-bit processor.
- MIPS has gone through several generations and is available in ___ 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 _____ and _____ applications and in some _____ _____ ; for example, _____.
- MIPS has a classic ___-bit _____ and _____ ISA and ___ _____-_____ registers. Register r0 is unusual because it holds a _____ and cannot be changed. This is an important feature of MIPS because it allows the programmer easy access to _____.

4.3 MIPS: Instruction Formats



4.3 MIPS: R-type and I-type Instructions

- The _____ instruction provides a _____-__-_____ data processing operation. The most significant difference between MIPS and ARM processors is that MIPS can specify one of ___ registers, whereas ARM provides only ___ registers.
- A typical _____ instruction is _____. MIPS lacks two important ARM processor mechanisms, _____ _____ and the ability to _____ the _____ _____.
- The _____ instruction has a __-bit _____ field to provide a _____ in operations like ____ _____ or an offset in _____ _____ addressing modes. The ___-bit literal which may be signed or unsigned permitting a range of _____ to +_____ or __ to _____. The literal cannot be scaled.
- A typical _____ operation is _____. MIPS appends an i to the _____ 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

4.3 MIPS: 32-bit Constants and J-type Instructions

- Because MIPS uses 16-bit literals, depositing a ___-bit word into a register is easily done by loading ___ _____ literals.
- A _____ _____ _____ instruction, _____, deposits a 16-bit literal into the _____-_____ 16-bits of a register and clears the _____-_____ 16 bits to zero; for example _____ loads register _____ with _____.
- A logical ___ with a 16-bit immediate operand can now be used to access the lower-order 16 bits; for example, _____ will set _____ to _____.
- The _____ instruction format is _____ _____ and provides a ___-bit literal that is used to construct a branch _____.
- 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.

4.3 MIPS: Registers, Load, Store, Addressing

- The MIPS register set is conventional and, apart from `r0` that is fixed at 0, has no _____-_____ registers .
- MIPS assembly language uses _____– … rather than _____ … as the name of registers.
- MIPS _____ and _____ instructions are ____ (load word) and ____ (store word). Addressing modes are minimal and MIPS provides only a _____ _____ with _____ addressing mode; for example _____ implements _____.
- 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.

4.3 MIPS: 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 _____ _____ _____ set or cleared by a previous instruction.
- MIPS provides _____ compare and branch instructions; for example, _____ compares the contents of register r1 with r2 and branches to _____ on _____.
- 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
```

4.3 MIPS: Set on Condition Instruction

- An interesting MIPS instruction is the set on condition; for example, the ___ __ ____ _____ instruction _____ performs the test _____ and then sets ____ to ___ if the test is true and to ___ 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
- There is also an `sltu` operation that performs the same operation on _____ numbers, and `slti` and `sltui` versions that have _____ operands.

4.3 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 _____ shift instructions that provide either a _____ length shift with a _____ shift field, or a _____ shift with a _____ shift field; for example,