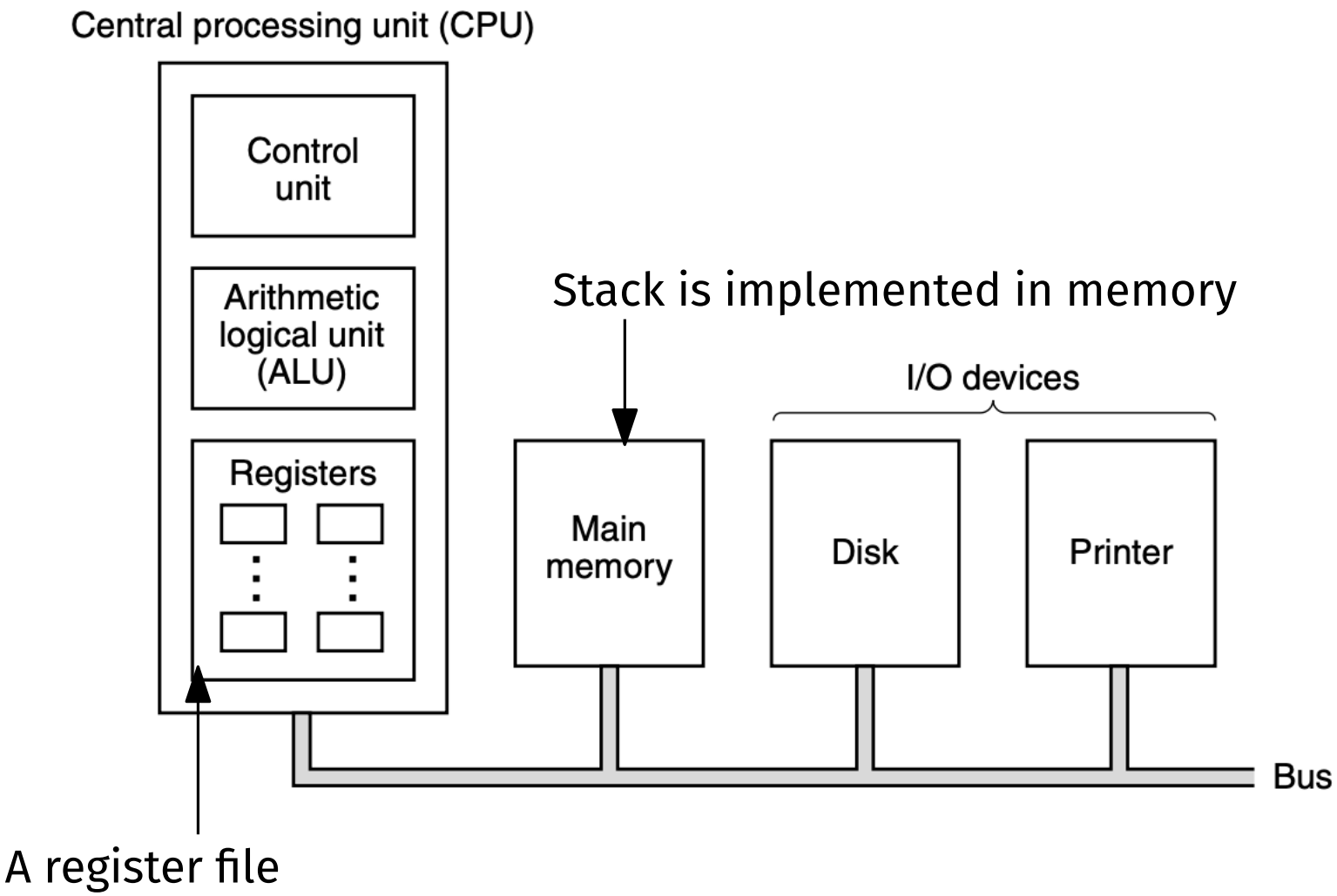
Modern General Purpose Register Machines

Modern processors use many techniques to optimize execution.

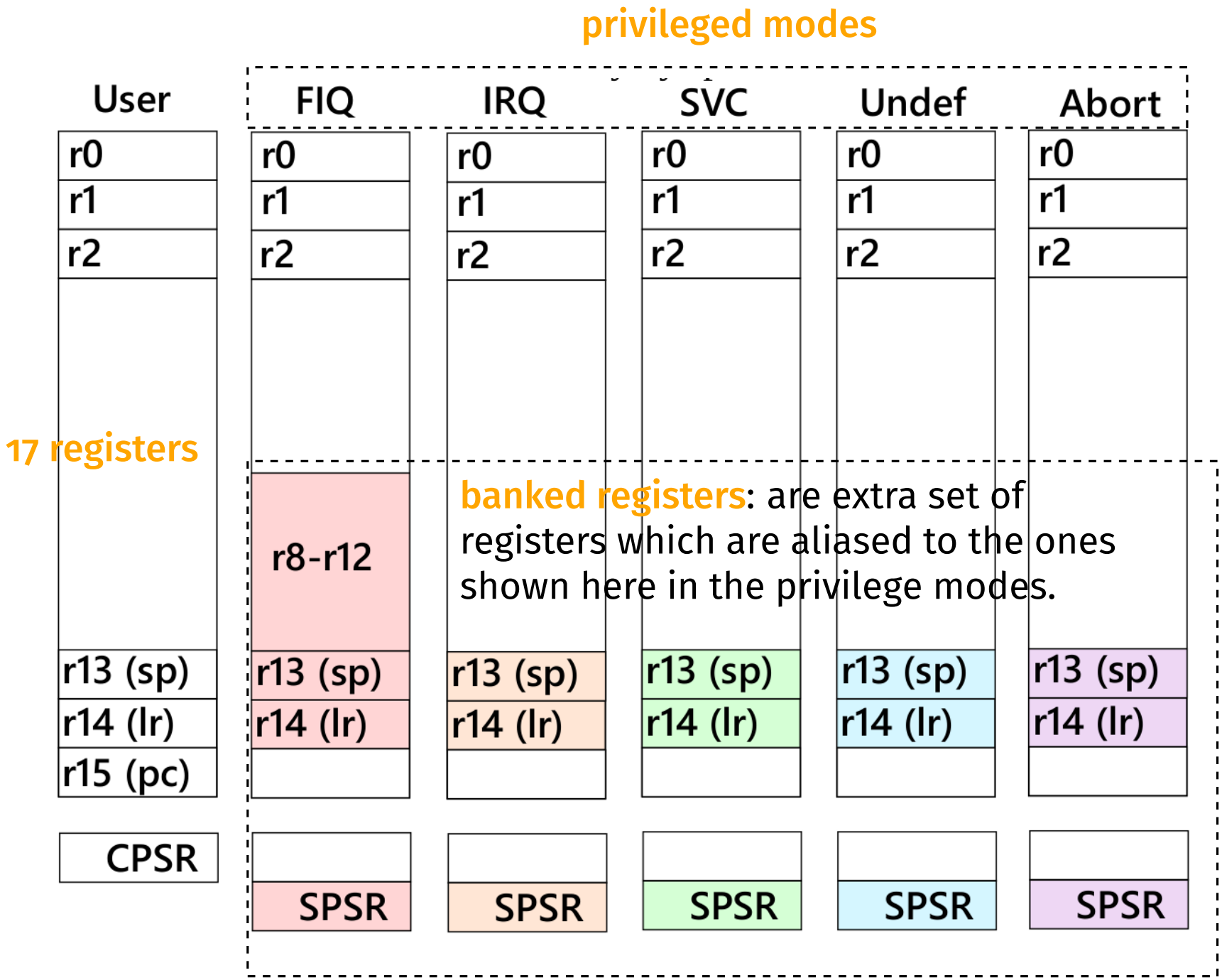


The ARM ISA

Architecture Overview

- there are 37 registers, out of which 17 are visible to users
- 15 of these are general purpose (r0-r14), r15 is PC and there is a CPSR (common processor status register)
- Instructions are 32 bit fixed with, 16 bit for Thumb ISA.
- 3 operand format is used
- Only load and store can access the memory
- all instruction can be executed conditionally
- Main memory can be accessed in byte-by-byte manner

ARM register set



ARM processors modes

Modes are used to implement access control to the hardware resources.

There are 7 modes, and they can be accessed via specific interrupts.
The interrupt handlers are located at the addresses shown below and have fixed priorities.

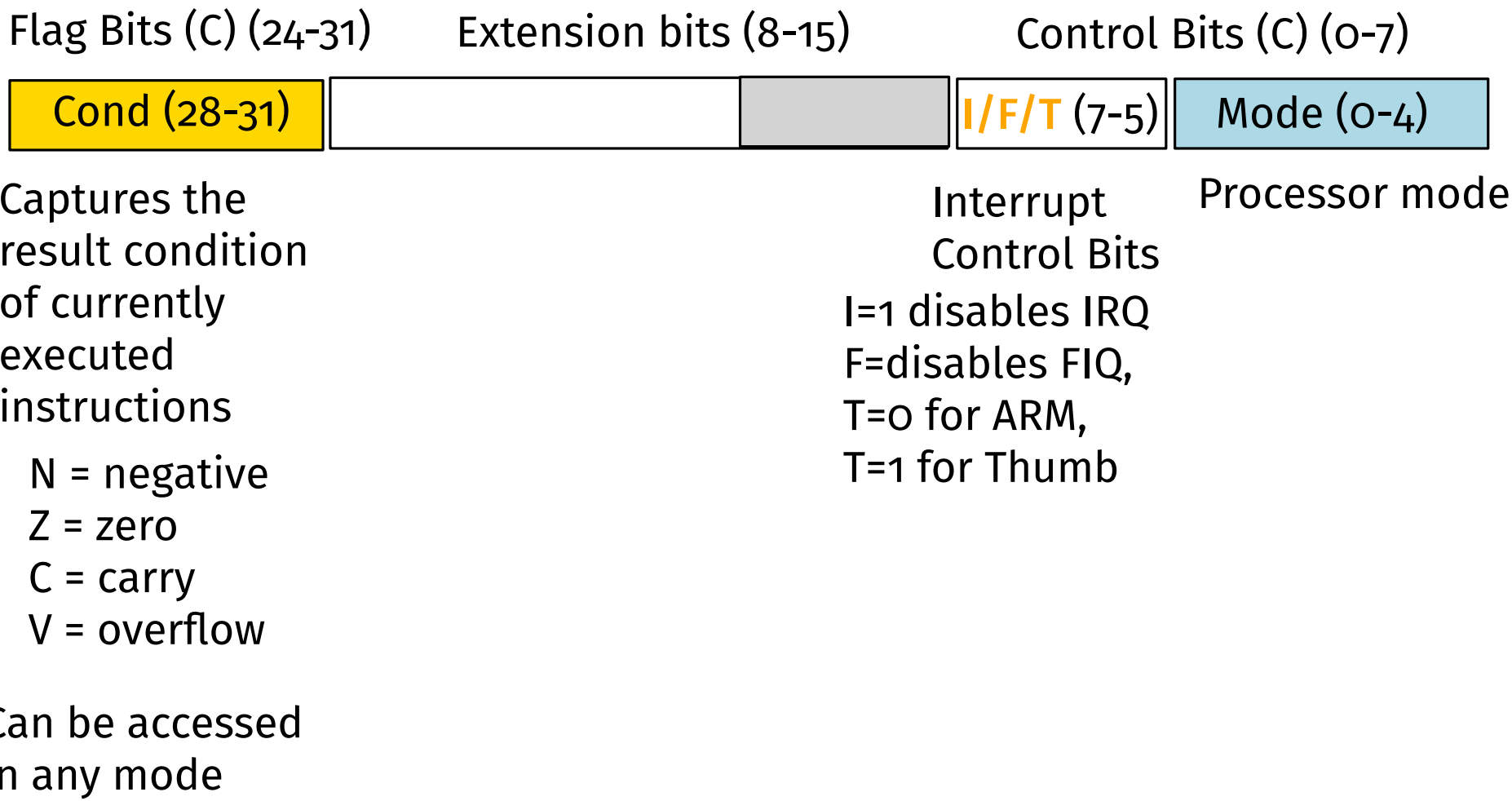
Mode	Address	Priority	Description
Undefined (undef)	0x4	6	for emulation of coprocessors
Supervisor (swi)	0x8	6	for operating system
Prefetch abort (abort)	0xC	5	instruction access problem
Data abort (abort)	0x10	2	memory management
Interrupt (irq)	0x18	4	regular interrupts
Fast Intr. (firq)	0x1C	3	fast interrupts (e.g. DMA)
System (sys)	-	-	used for reentrant interrupts

Privilege modes can read and write to status registers and can access memory at any location.

System mode (sys) is a hybrid of Supervisor and User modes - it has privileges of supervisor mode, but uses user mode registers. Can be accessed only through CPSR bits.

Processor Status Registers

CPSR has following bits



Manipulation of PSRs

To copy a register into a PSR:

```
MSR     CPSR, R0            ; Copy R0 into CPSR
MSR     SPSR, R0            ; Copy R0 into SPSR
MSR     CPSR_f, R0          ; Copy flag bits of R0 into CPSR
MSR     CPSR_f, #1<<28      ; Copy flag bits (immediate) into CPSR
```

and to copy from a PSR:

```
MRS     R0, CPSR            ; Copy CPSR into R0
MRS     R0, SPSR            ; Copy SPSR into R0
```

Changing mode is done for example:

```
MRS     R0, CPSR            ; Copy the PSR
BIC     R0, R0, #&1F        ; Clear the mode bits
ORR     R0, R0, #new_mode   ; Set bits for new mode
MSR     CPSR_c, R0          ; write PSR back with control bits
```

Data Processing Instructions

Arithmetic	Logicals	Comparisons	Data Movement Between Regs
addition subtraction multiplication	And, or.. Shifts Rotations	cmp, cmn tst teq	mov, mvn

- These operations can have 3 operands: 2 source and 1 destination
- Only registers or immediate values can be operands
- Second operand can be passed through a barrel shifter
- Can be executed conditionally
- Can set the condition code in CPSR

Examples -

add r0, r1, r2 r0 = r1 + r2

sub r0, r1, r2 r0 = r1 - r2

and r0, r1, r2 r0 = r1 & r2

Instruction Coding

Instructions are coded to identify the operation, the operands (3), conditions and shifts.

	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
Data processing immediate shift	cond [1]	0 0 0	opcode	S	Rn	Rd	shift amount	shift	0	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	x x x	0	x x x x
Data processing register shift [2]	cond [1]	0 0 0	opcode	S	Rn	Rd	Rs	0 shift	1	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	0 x x	1	x x x x
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0 0 0	x x x x	x	x x x x	x x x x	x x x x	1 x x	1	x x x x
Data processing immediate [2]	cond [1]	0 0 1	opcode	S	Rn	Rd	rotate	immediate		
Undefined instruction [3]	cond [1]	0 0 1	1 0 x	0 0	x x x x	x x x x	x x x x	x x x x	x x x x	
Move immediate to status register	cond [1]	0 0 1	1 0 R	1 0	Mask	SBO	rotate	immediate		
Load/store immediate offset	cond [1]	0 1 0	P U B	W L	Rn	Rd	immediate			
Load/store register offset	cond [1]	0 1 1	P U B	W L	Rn	Rd	shift amount	shift	0	Rm
Undefined instruction	cond [1]	0 1 1	x x x x	x	x x x x	x x x x	x x x x	x x x	1	x x x x
Undefined instruction [4,7]	1 1 1 1	0 x x x	x x x x	x	x x x x	x x x x	x x x x	x x x	x x x x	
Load/store multiple	cond [1]	1 0 0	P U S	W L	Rn	register list				
Undefined instruction [4]	1 1 1 1	1 0 0	x x x x	x	x x x x	x x x x	x x x x	x x x	x x x x	
Branch and branch with link	cond [1]	1 0 1	L	24-bit offset						
Branch and branch with link and change to Thumb [4]	1 1 1 1	1 0 1	H	24-bit offset						
Coprocessor load/store and double register transfers [6]	cond [5]	1 1 0	P U N	W L	Rn	CRd	cp_num	8-bit offset		
Coprocessor data processing	cond [5]	1 1 1 0	opcode1		CRn	CRd	cp_num	opcode2	0	CRm
Coprocessor register transfers	cond [5]	1 1 1 0	opcode1	L	CRn	Rd	cp_num	opcode2	1	CRm
Software interrupt	cond [1]	1 1 1 1	swi number							
Undefined instruction [4]	1 1 1 1	1 1 1 1	x x x x	x	x x x x	x x x x	x x x x	x x x	x x x x	

Conditional execution

All instructions can be executed conditionally:

add[cc][s] r0, r1, r2 addeq r0, r1, r2

condition code refers CPSR condition bits from previous execution

Mnemonic	Condition	Mnemonic	Condition
CS	*C*arry *S*et	CC	*C*arry *C*lear
EQ	*E*qual (Zero Set)	NE	*N*ot *E*qual (Zero Clear)
VS	*O*verflow *S*et	VC	*O*verflow *C*lear
GT	*G*reater *T*han	LT	*L*ess *T*han
GE	*G*reater Than or *E*qual	LE	*L*ess Than or *E*qual
PL	*P*lus (Positive)	MI	*M*inus (Negative)
HI	*H*igher Than	LO	*L*ower Than (aka CC)
HS	*H*igher or *S*ame (aka CS)	LS	*L*ower or *S*ame

above instruction will be executed when Z bit is set in CPSR from previous results

Conditional execution

Used to reduce code size, improve processor performance. Compare:

```
cmp r0, #0
BEQ label1
add r1, r2, r3
label1: ...
```

→

```
cmp r0, #0
addne r1, r2, r3
...
```

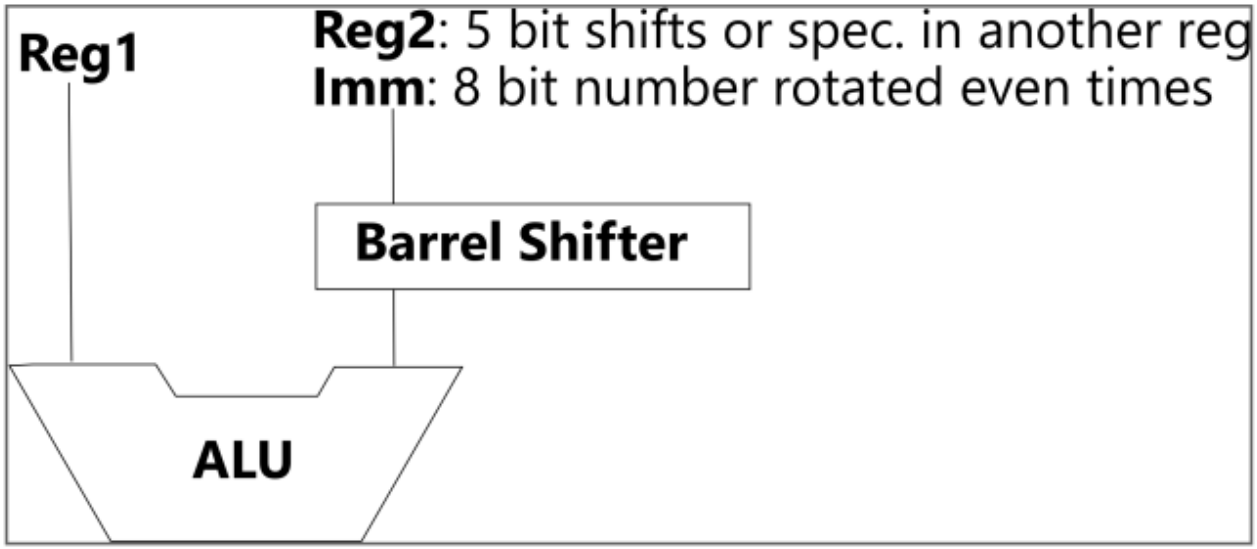
```
cmp r0, #2
beq L1
add r1, r1, r0
mul r2, r1, r2
L1: ...
```

→

```
cmp r0, #2
addne r1, r1, r0
mulne r2, r1, r2
```

The Barrel Shifter

The second operand can be passed through a barrel shifter for free. This allows implementation of shifts as well as efficient arithmetic operations.



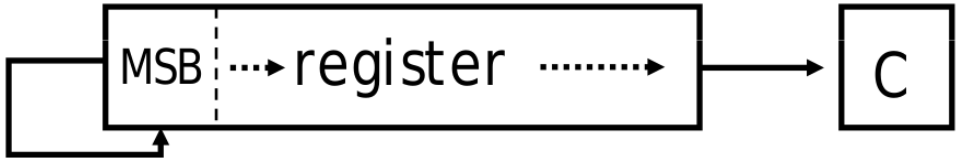
Mnemonic	Description	Shift	Result
LSL	logical shift left	xLSL y	$x \ll y$
LSR	logical shift right	xLSR y	$(\text{unsigned})x \gg y$
ASR	arithmetic right shift	xASR y	$(\text{signed})x \gg y$
ROR	rotate right	xROR y	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$
RRX	rotate right extended	xRRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$

Various shift operations

logical shift left
add r0, r2, r2, lsl #1

logical shift right
add r0, r2, r2, lsr #1

arithmetic shift right
add r0, r2, r2, asr #1



rotate right
add r0, r2, r2, ror #1

rotate right extended
add r0, r2, r2, rrx #1

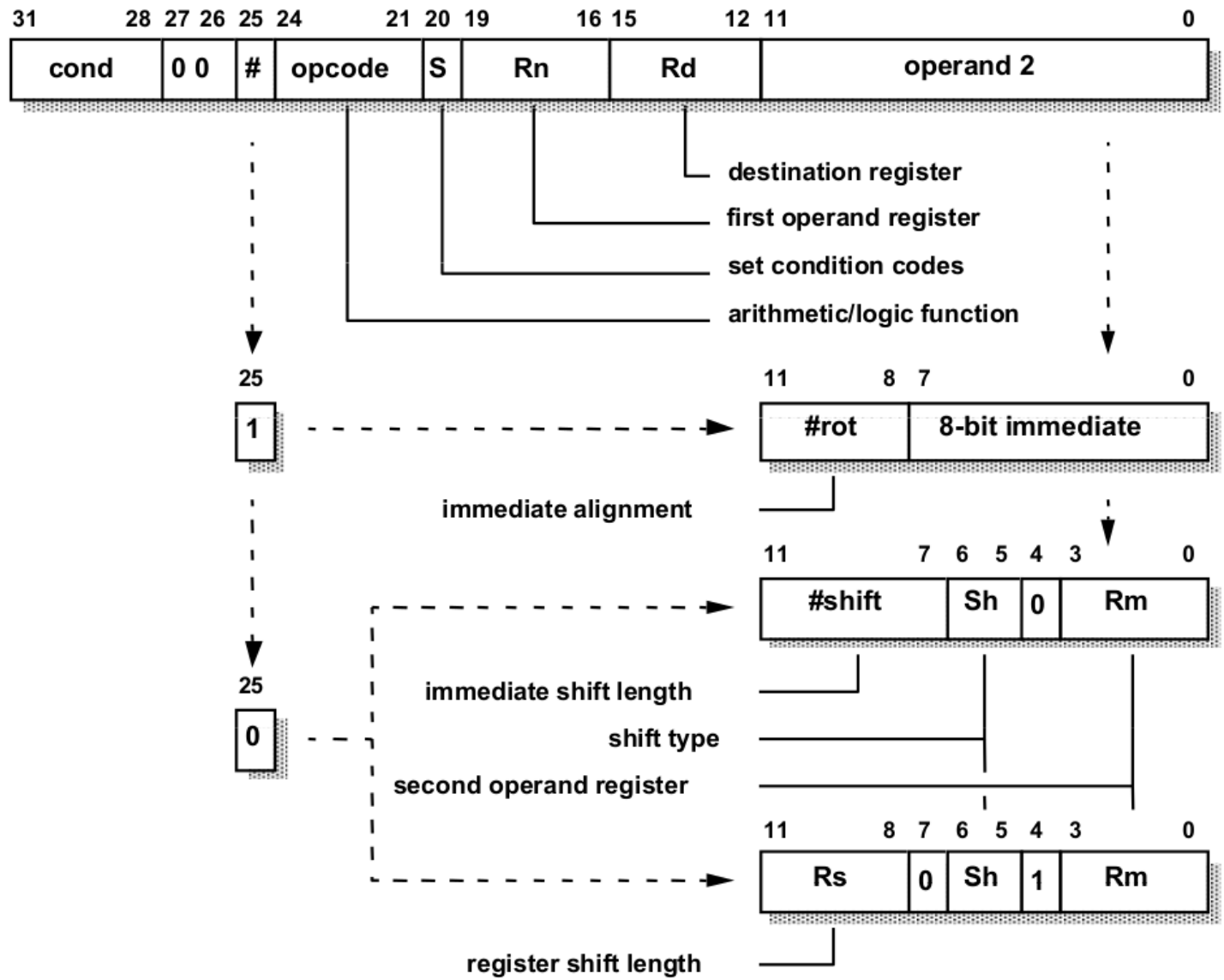
How shifts are encoded

When second operand is a register

- can be inside the instruction (5 bits): add r0, r1, r2, lsl #3
- can be from botton 8 bits of another register: add r0, r1, r2, lsl r4

When second operand is an immediate value

- 8 bits for immediate value and 4 bits for shift type: add r0, r1, r2, lsl #1



Some uses of shifts

Fast multiply by many small constants -

```
// r2 = 45 x r0
add r0, r0, r0, LSL #2 //r0 = 5 x r0
add r2, r0, r0, LSL #3 //r2 = 9 x r0
```

Instead of -

```
mov r1, #45
mul r2, r0, r1
```

Arithmetic operations

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

How to do 64-bit addition?

r2 = r2 + r0
r3 = r3 + r1 // with carry from previous sum

```
adds r2, r2, r0
adc r3, r2, r1
```

Logical operations

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Bit clear operation clears the bits of r1, that are specified in r2, and stores the result in r0.

```
bic r0, r1, r2
```

Comparison operations

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

No results are generated from these operations, only the NZCV bits of CPSR are updated.

Multiplications

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	multiply	$Rd = Rm * Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

Second operand can not be an immediate value.

The first operand should not be same as result register.

Branching instructions

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

Like C's goto statement

Used for function calls

exchanges ARM and Thumb instructions

B	branch	*pc* = *label* pc-relative offset within 32MB
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL (R14)
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

Branching conditions

Mnemonic	Name	Condition flags
EQ	equal	Z
NE	not equal	z
CS HS	carry set/unsigned higher or same	C
CC LO	carry clear/unsigned lower	c
MI	minus/negative	N
PL	plus/positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater than or equal	NV or nv
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (unconditional)	ignored

Examples

Loops: consider the following:

```
    mov    r0, #0
    mov    r1, #0
loop
    cmp    r0, #10
    bge    fin
    add    r1, r1, r0
    add    r0, r0, #1
    b    loop
fin
```

What does this calculate?

How do we test it? Can this code be compiled and run?

Consider a simple C program (a.c):

```
int main(){
    int i=0;
    return 0;
}
```

get assembly using:

```
cc -S a.c
```

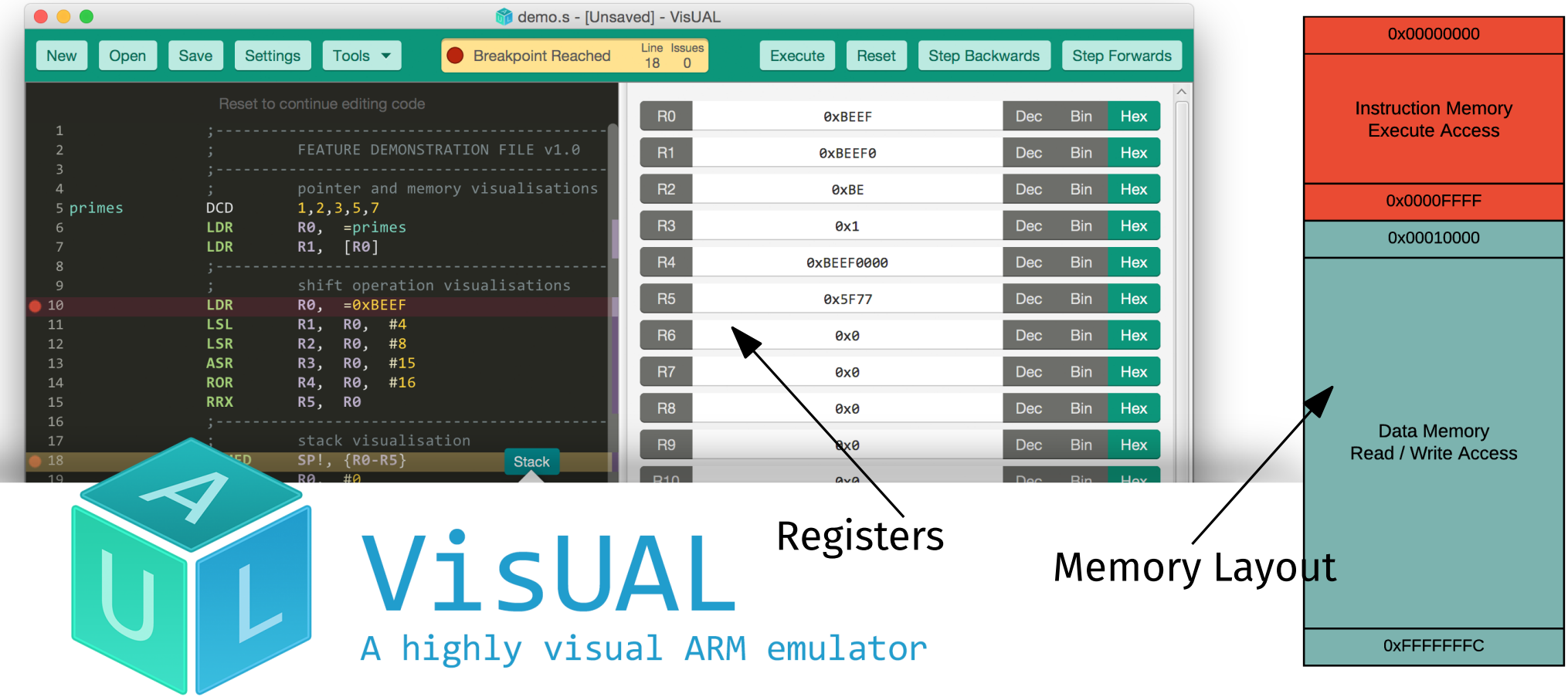
which gives a.S

contains lots of stuff for coordination with the operating system!

```
    .section    __TEXT,__text,regular,
      pure_instructions
    .build_version macos, 11, 0
      sdk_version 11, 1
    .globl    _main                    ; --
      Begin function main
    .p2align   2
_main:
      ; @main
    .cfi_startproc
; %bb.0:
    sub sp, sp, #16                    ; =16
    .cfi_def_cfa_offset 16
    mov w8, #0
    str wzr, [sp, #12]
    str wzr, [sp, #8]
    mov x0, x8
    add sp, sp, #16                    ; =16
    ret
    .cfi_endproc

      ; -- End function
```

VISUAL: ARM emulator

The supported instructions are listed at:
https://salmanarif.bitbucket.io/visual/supported_instructions.html



Several visualisation are available: Pointers, Memory Access, Shift operations, Stack use.

ARM ISA: 64 Bit addition

```
;     64 bit addition
;     v1:  12A2E640,F2100123
;     v2:  001019BF,40023F51
;     sum:  12B30000,32124074

      adr   r0, val1
      ldmia r0, {r1, r2}
      adr   r0, val2
      ldmia r0, {r3, r4}
      adds    r6, r2, r4
      adc   r5, r1, r3
      adr   r0, result
      stmia r0, {r5, r6}

val1    dcd   0x12A2E640, 0xf2100123
val2    dcd   0x001019bf, 0x40023f51
result  dcd   0x0
```

Comments

Load and store instructions

directive to allocate one or more words of memory, aligned on four-byte boundaries

ARM ISA: Lookup table example

```
;   lookup table for factorials.
;
adr   r10, data
adr   r11, value
ldr   r1, [r11, #0]
mov   r1, r1, lsl #0x2
add   r10, r10, r1
ldr   r2, [r10]
adr   r3, result
str   r2, [r3]

data    dcd   1, 1, 2, 6, 24, 120, 720, 5040
value dcd   6
result  dcd   0x0
```

psuedo-instruction for getting address of a label

Factorial is not calculated but obtained from a list.

ARM ISA: Function Definition

Branching instructions are also used for calling functions. In this case you need to return back to the instruction where the call was made.

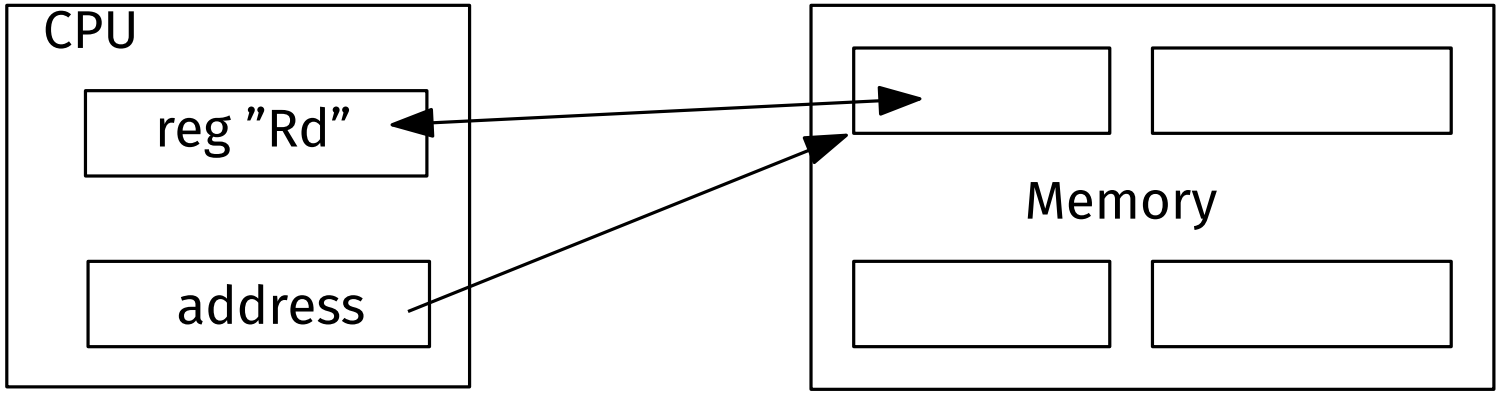
```
main
    adr   r1, hdigit
    ldr   r0, [r1]
    bl    func
    adr   r1, result
    str   r0, [r1]
    bl    stop
func
    ;   adds 2 if r0 <= 0xa
    ;   adds 3 if r0 > 0xa
    cmp   r0, #0x0a
    ble   next
    add   r0, r0, #0x1
next
    add   r0, r0, #0x2
    mov   pc, lr

hdigit  dcd   0xb
result  dcd   0x0

stop
```

branch and "link" to this location

Load and Store instructions

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



Addressing modes

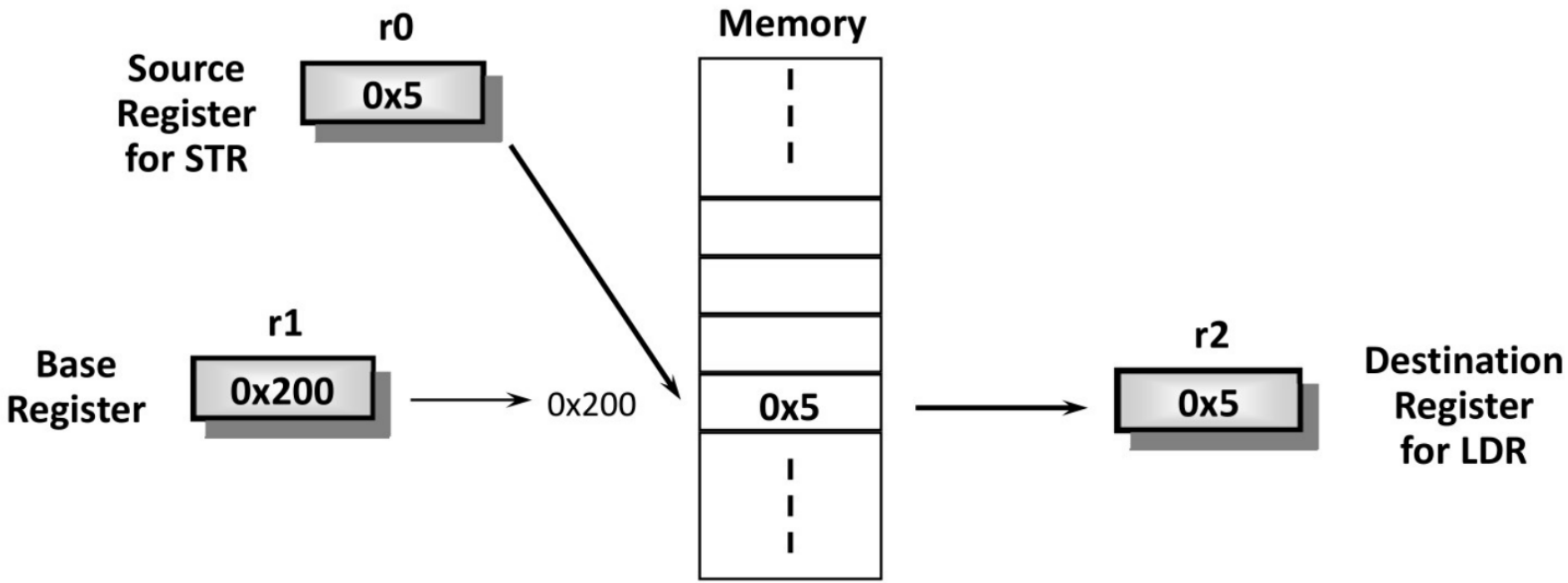
```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	[Rn, #+/-offset_12]
Preindex with register offset	[Rn, +/-Rm]
Preindex with scaled register offset	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with scaled register offset	[Rn, +/-Rm, shift #shift_imm]!
Immediate postindexed	[Rn], #+/-offset_12
Register postindex	[Rn], +/-Rm
Scaled register postindex	[Rn], +/-Rm, shift #shift_imm

Load/Store addressing mode: displacement

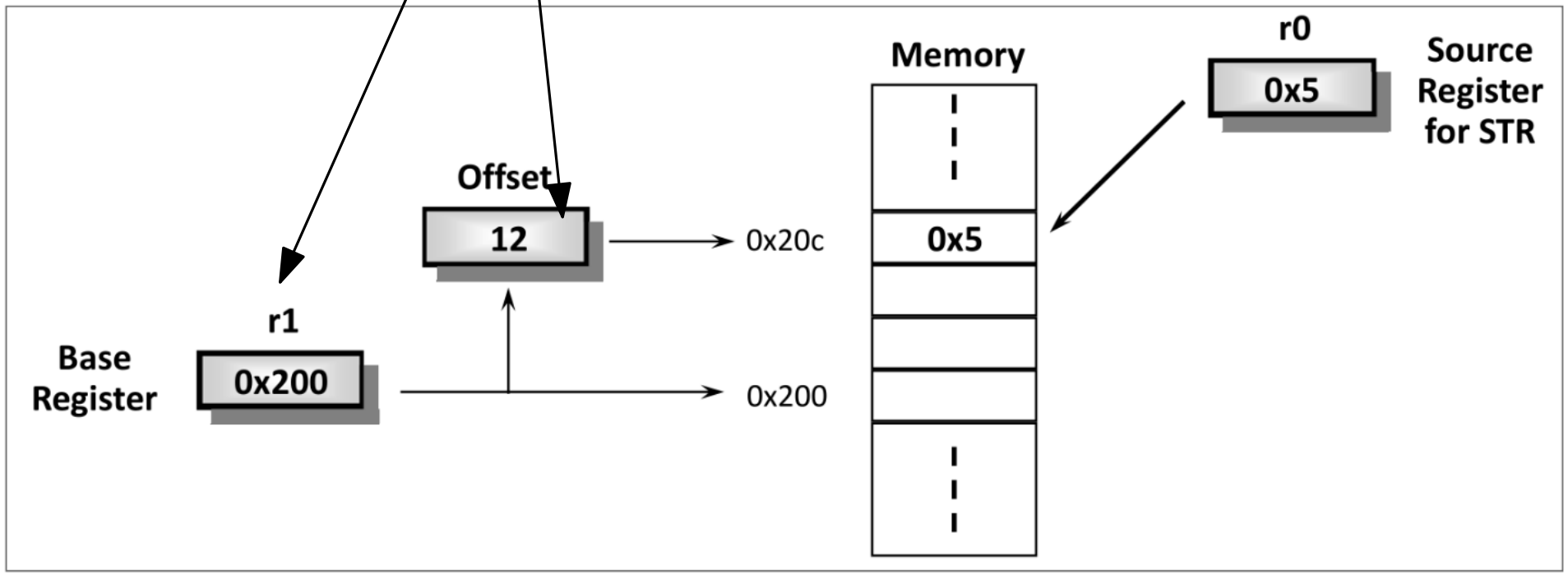
- **str r0, [r1]**
- **ldr r2, [r1]**

Basic mode load and store



Load/Store addressing mode: displacement

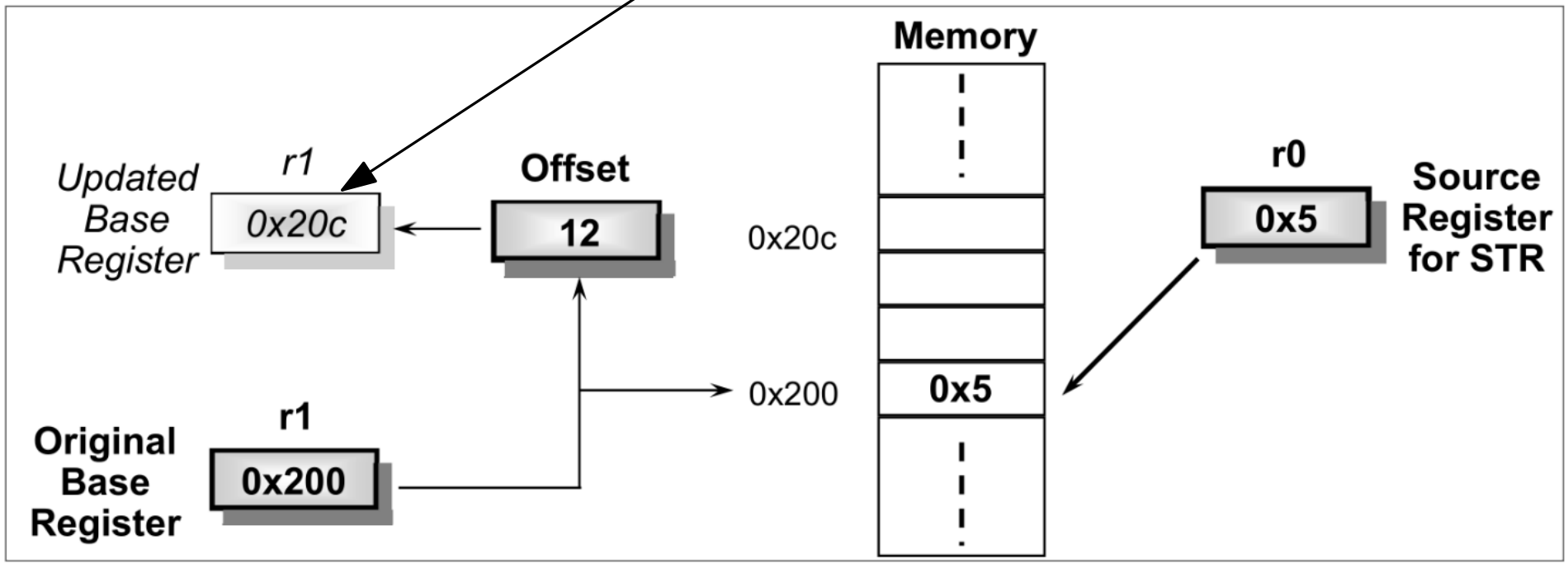
- 12 bit offset : **str r0, [r1,#12]** Is this 12 bit offset useful?



- a register optionally shifted: **str r0, [r1, r2, lsl#2]**
  does same as above if r2 contains 3.

Load/Store addressing mode: displacement with increments

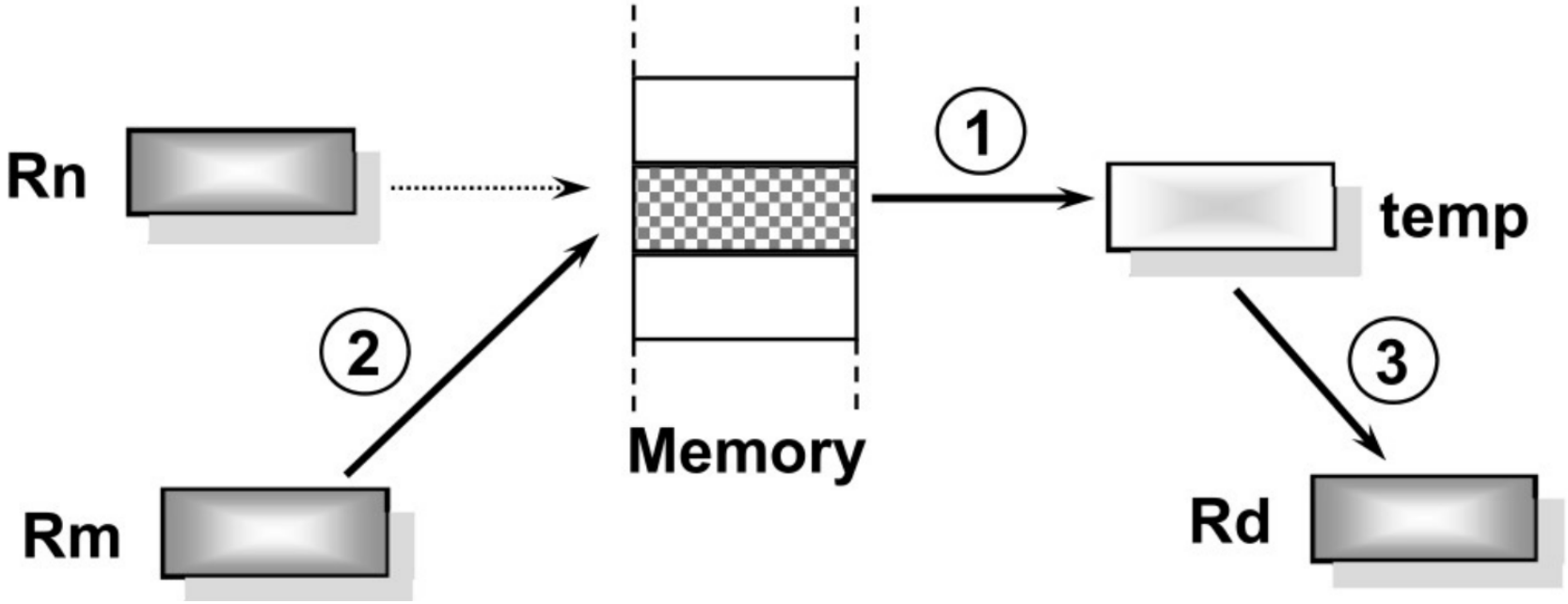
- pre-indexed addressing: **str r0, [r1,#12]!**
- post-indexed addressing: **str r0, [r1], #12**



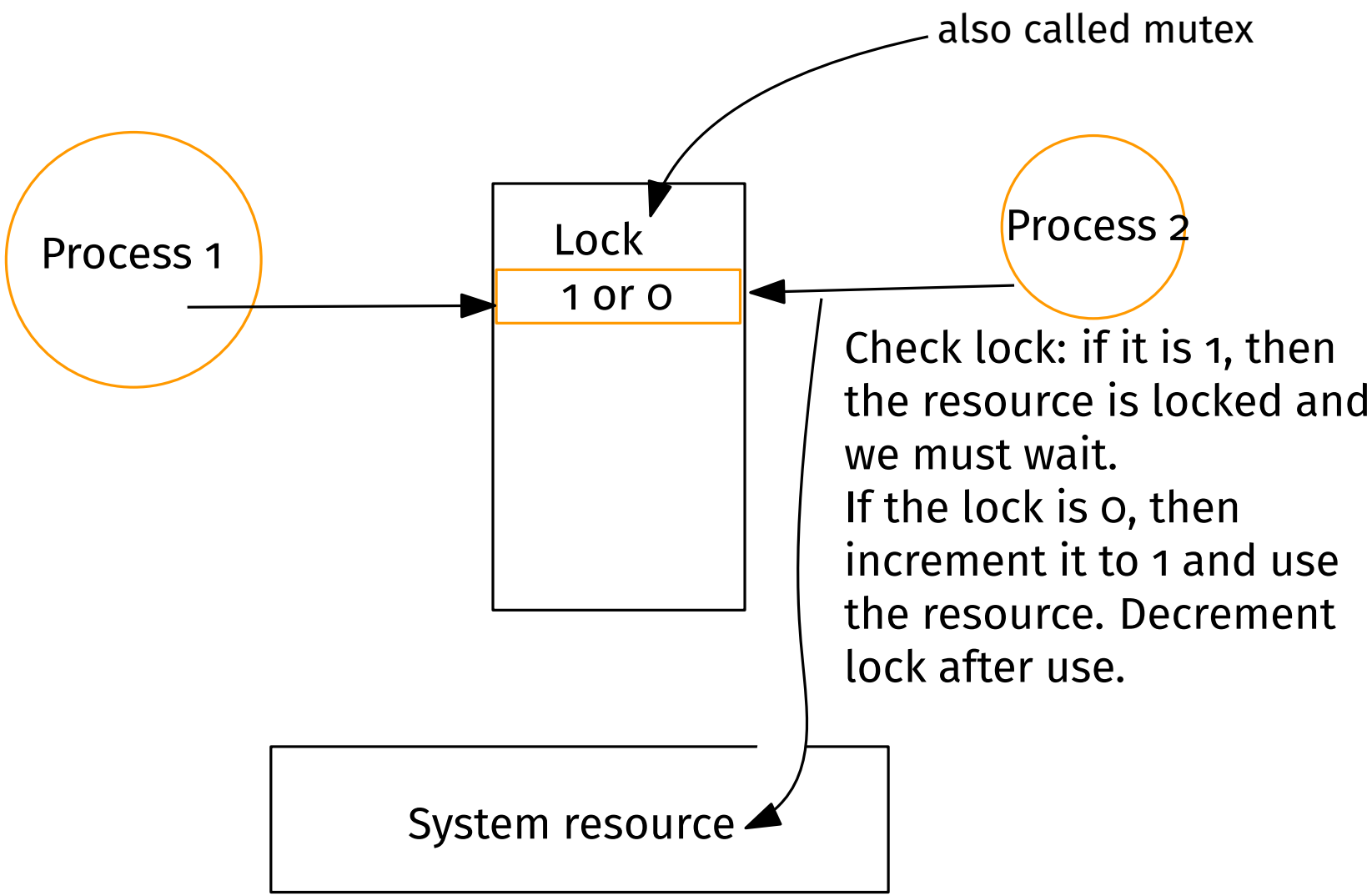
- or: **str r0, [r1], r2, lsl#2**
  does same as above if r2 contains 3.
- used in array access

Atomic Byte Swaps

- Atomic operation: moves bytes between registers | memory
- used for locks, securing operations
- **swp[cond][s] rd, rm[, rn]** ← None of these registers is PC



Atomic Byte Swaps for locking



What happens in multiple core microprocessors?

Atomic Byte Swaps for locking

How do we implement this?

Lock is stored in [R1]

```
try: MOV R0,#1
     SWP R2,R0,[R1,#0]
     CMP R2,#1
     BEQ try
// can access the resource here
Do work
// resource usage finished
MOV R0,#0
SWP R2, R0, {R1,#0}
```

Set lock to 1 and also check if it was 1 before.

Decrement lock to 0

Can this work in presence of malicious users?

Load / Store Exclusive

Locks can also be created using LDREX and STREX, which provide exclusive access to memory locations.

ldrex[cond], Rd, [Rs [, #offset]]
strex[cond], Rd, Rs1 [RS2, #offset]]

For STREX, Rd gets the status of the operation
0 for success, 1 for non-success

LDREX and STREX should be paired and refer to the same memory location.

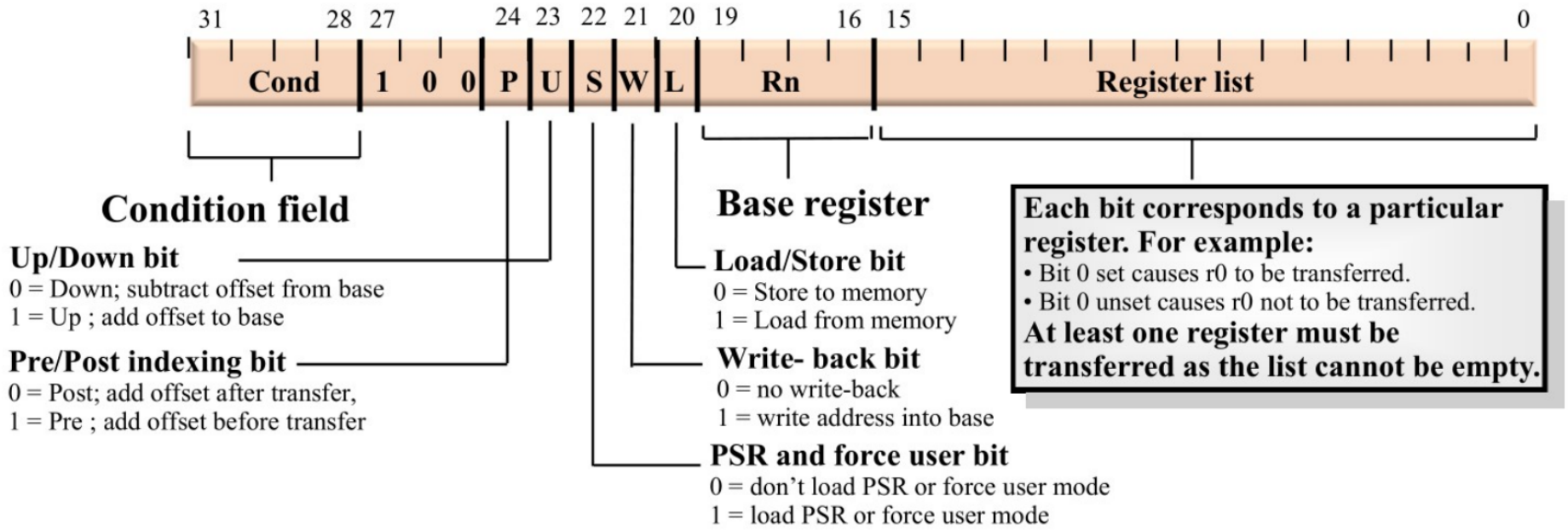
```
MOV r2, #1
try
    LDREX r0, [r1]
    CMP r0, #0
    STREXEQ r0, r2, [r1]
    CMPEQ r0, #0
    BNE try

// resource can be used here
do work

// release the lock, how?
```

[r1] is marked for exclusive access

Checks to see if STREX had success **why?**

Load/Store multiple registers

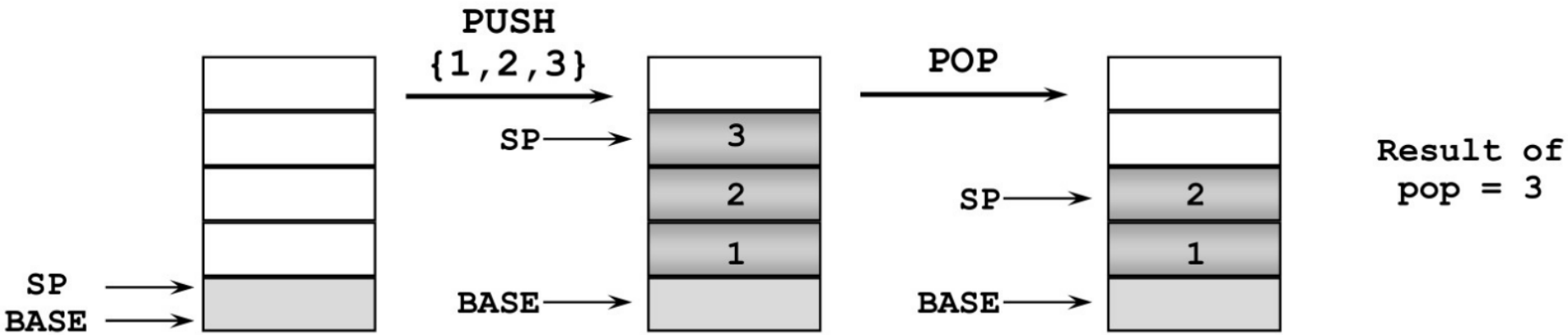
- **ldm**, **stm**: **stmfd sp!,{regs,lr}**
- **ldm[ia|ib|da|db]** or **stm[ia|ib|da|db]**
- 1 to 16 registers can be moved to/from memory by one instruction



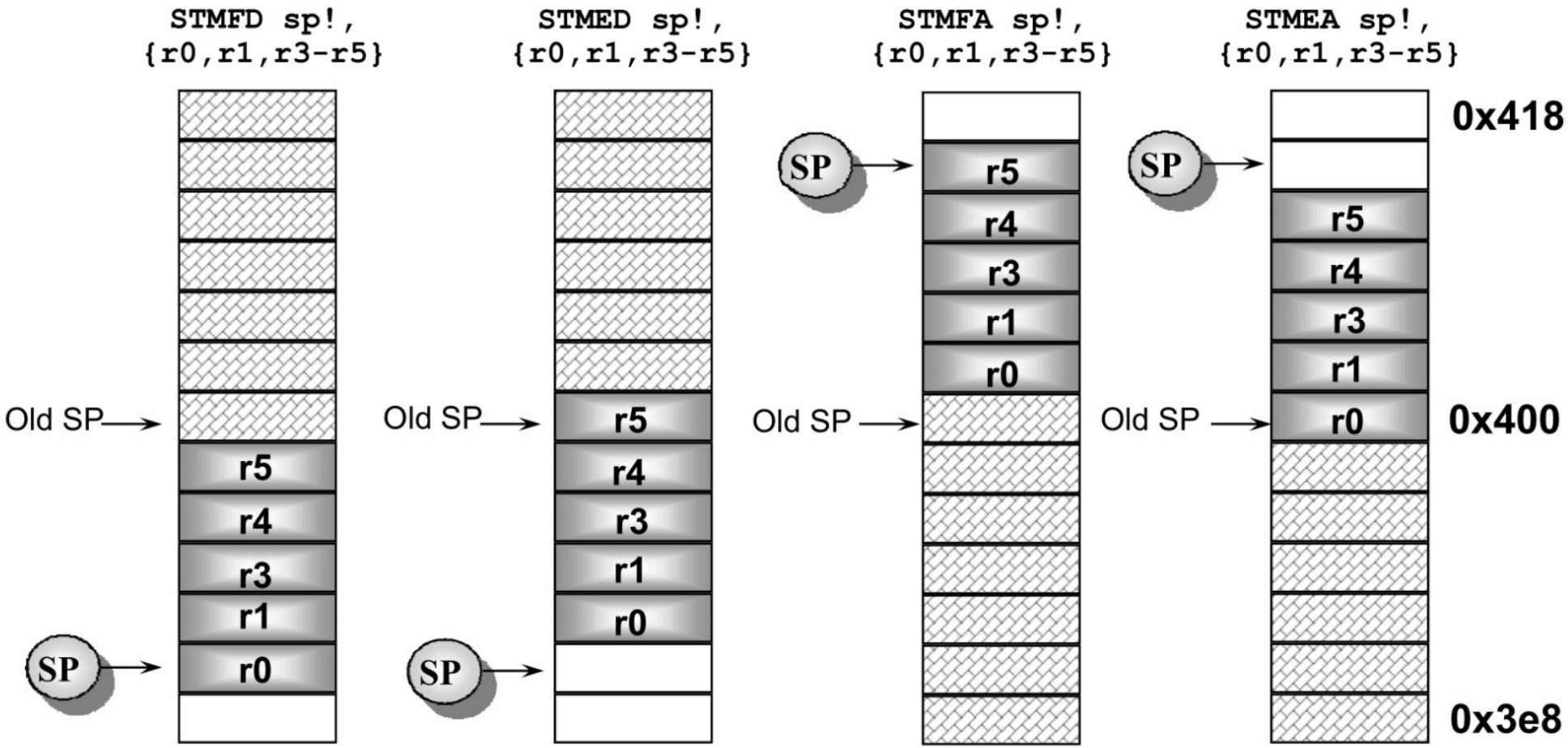
- Base register is used for memory address (normally on stack)
- Lowest register is transferred to lowest memory location
- used for
  - saving context (e.g. in function calls)
  - memory movements

Aside: Stack Operation

- Base pointer and stack pointer
- Stack pointer points to last occupied full address (full descending "FD" stack)
  - needs pre-decrement before push (on a *normal* stack)

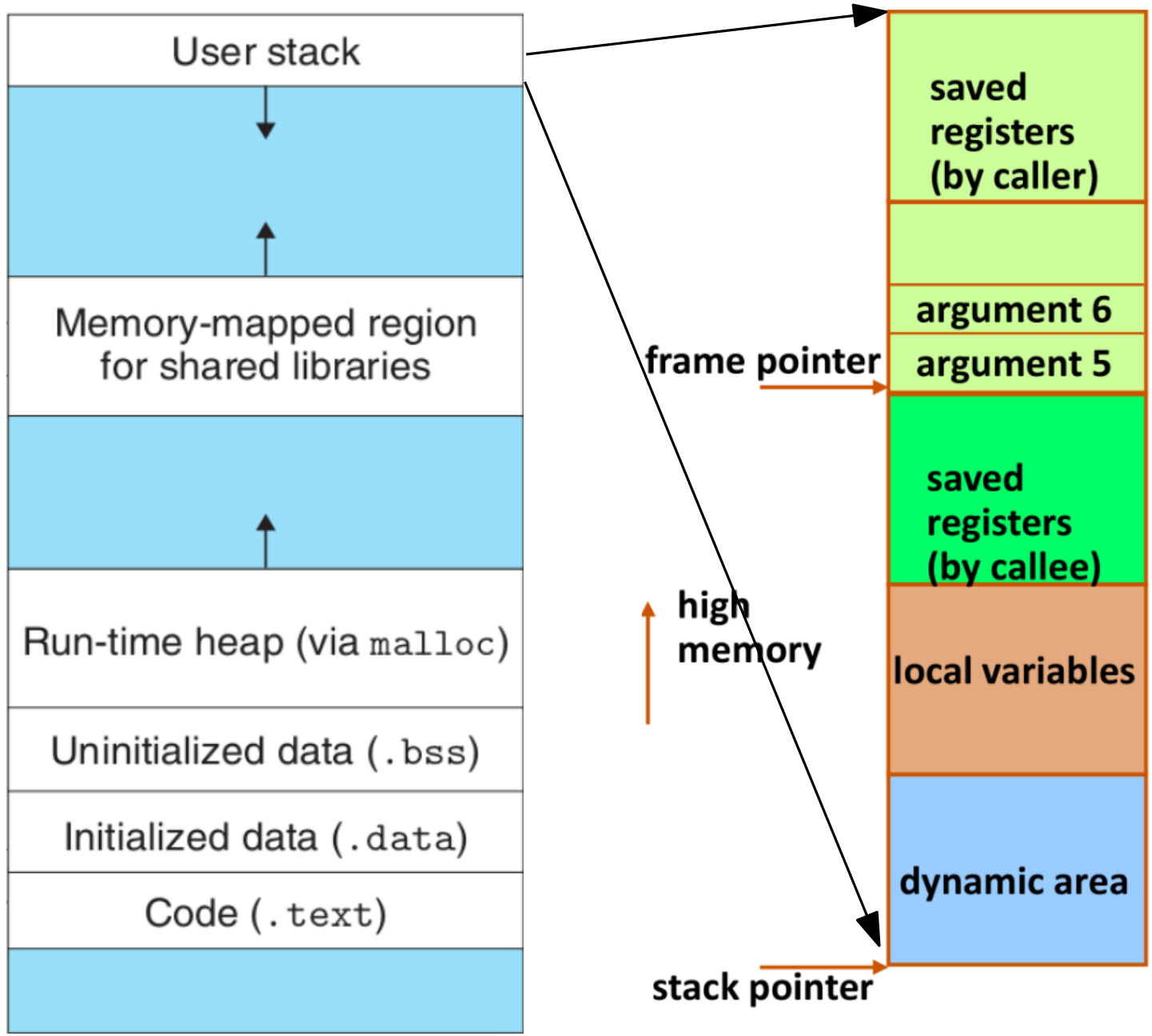


Stack Operation Examples With multi-Load/Store



Program layout in memory

A program can be assumed to have a linear and continuous memory allocated to it.
Here is how it looks like when a function call is made.



Function calling conventions

Suppose the following code is compiled into a program.

```
int add1(int x) { // callee
    int y; //local variables
    return y = x + 1;
}
int main(int argc, char **argv) { // caller
    int i = 0;
    printf("%d\n", add1(i));
    return 0;
}
```

Following steps are taken when the program executes:

- Store arguments for callee to access = fill in registers or stack.
- Call callee = change PC to point to callee.
- Callee acquires storage for local variables on stack and stores any registers it may be changing.
- Callee performs its function.
- Callee puts results in registers or on stack for caller to access.
- Callee restores saved registers. -
- Callee returns control to caller by changing PC using link register or PC value stored on stack.

ARM ISA: actual function definition

```
main
    push    {ip, lr}
    mov     r0, #1
    mov     r1, #3
    mov     r2, #-4
    bl      do_something
    mov     r1, r0
    ldr     r0, =output_str
    bl      printf
    mov     r0, #0
    pop     {ip, pc}

do_something
    push    {r4, lr}
    add     r4, r0, r1
    mov     r0, r2
    bl      abs
    add     r0, r4, r0
    pop     {r4, pc}
```

store registers which are going to be overwritten

restore the registers back to their original condition

ARM Procedure Call Standard (ACPS)

The function calling process is standardised in ACPS.

registers	name	intended function	Notes
r0-r3	**a0-a3**	argument passing, a0 for integer results	Caller saved if needed
r4-r8	**v1-v5**	register variable	should return unchanged, callee saves else
r9,r10	**sb/v6,sl/v7**	stack base, stack limit	"" ""
r11	**fp**	frame pointer	Can be used as scratch if saved properly
r12	**ip**	scratch/ new sb	"" ""
r13	**sp**	Lower end of current stack frame	"" ""
r14	**lr**	Link register	"" ""
r15	**pc**	Program counter	"" ""

Software Interrupt

SWI is used to call OS functionality (system calls) which run at higher privilege mode (SVC). SWI_number is passed to the interrupt service routine to determine which system call to serve. The argument for the syscall are located in standardized locations (registers or stack).

```
Syntax: SWI{<cond>} SWI_number
```

SWI is called SVC now.

SWI	software interrupt	$lr\_svc =$ address of instruction following the SWI $spsr\_svc = cpsr$ $pc =$ vectors + 0x8 $cpsr$ mode $= SVC$ $cpsr\ I = 1$ (mask IRQ interrupts)