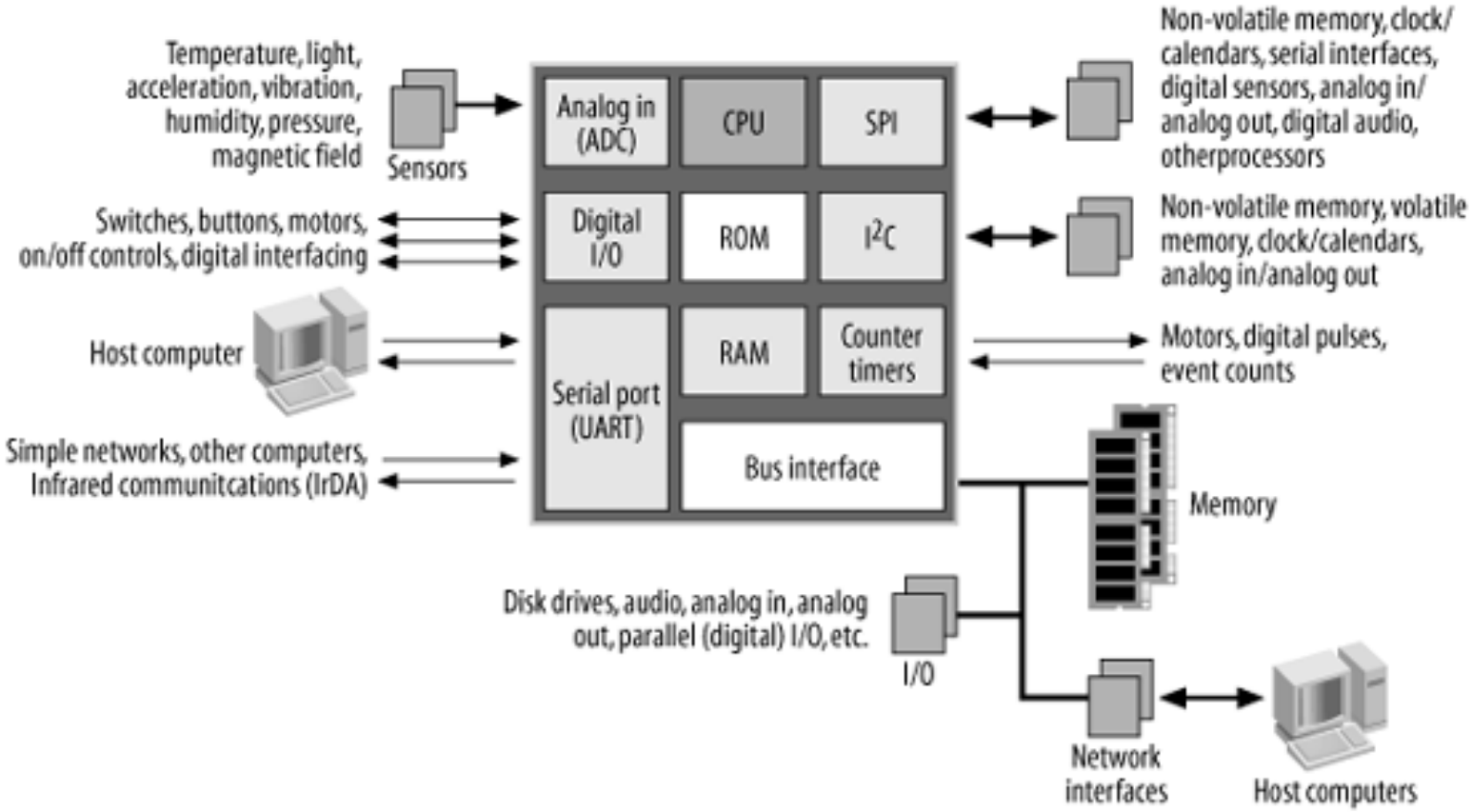
ARM Instruction Set

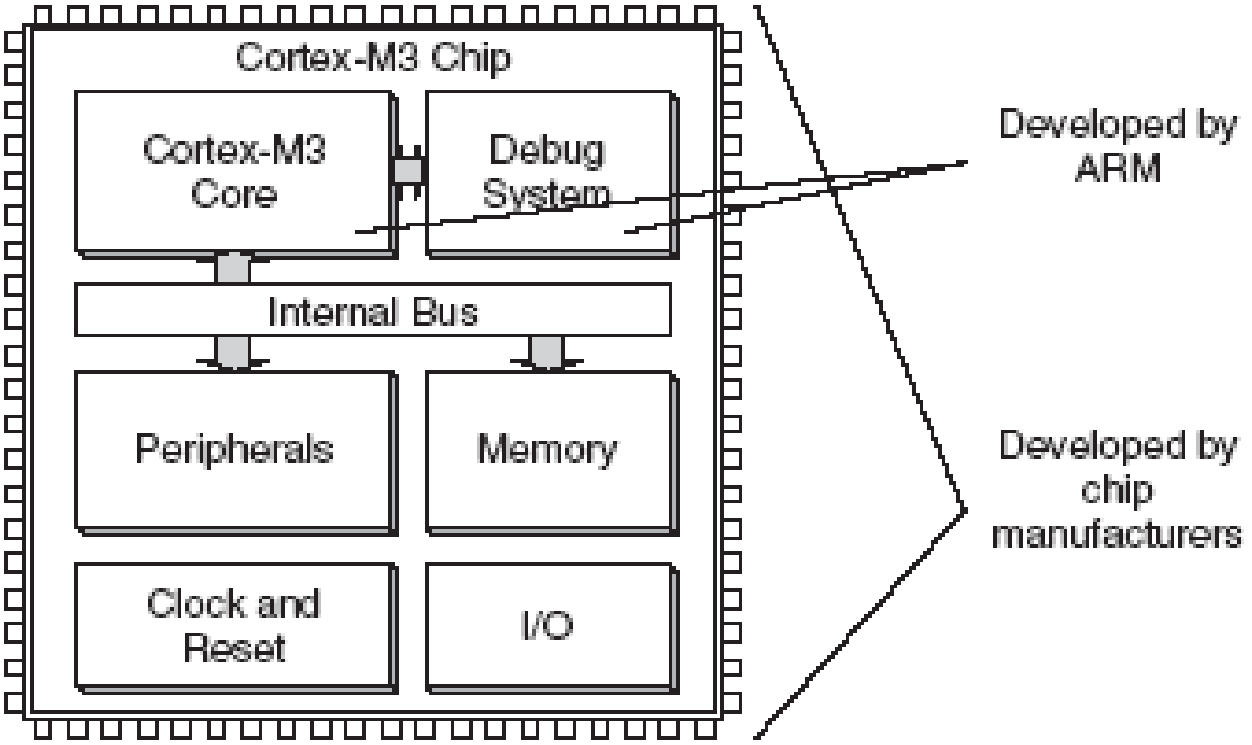
What is an embedded system?

- Components:
  - Processor(s)
  - Co-processors (graphics, security)
  - Memory (disk drives, DRAM, SRAM, CD/DVD)
  - input (mouse, keyboard, mic)
  - output (display, printer)
  - network
- Our primary focus on this chapter: the processor of ARM7 (datapath and control)
  - implemented using millions of transistors
  - Impossible to understand by looking at each transistor
  - We need...

Block diagram of Embedded Systems



ARM Cortex-M3 Microcontroller(MCU)



Choosing processor

- Existing processor

  - PIC, Intel X86, Motorola 68xx

- IP based Design
  - ARM

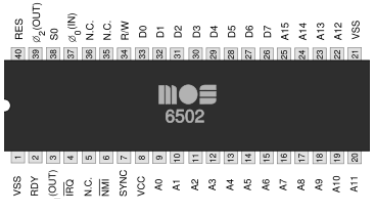
- Custom Design
  - Your own Instruction set (ISA)

ARM history

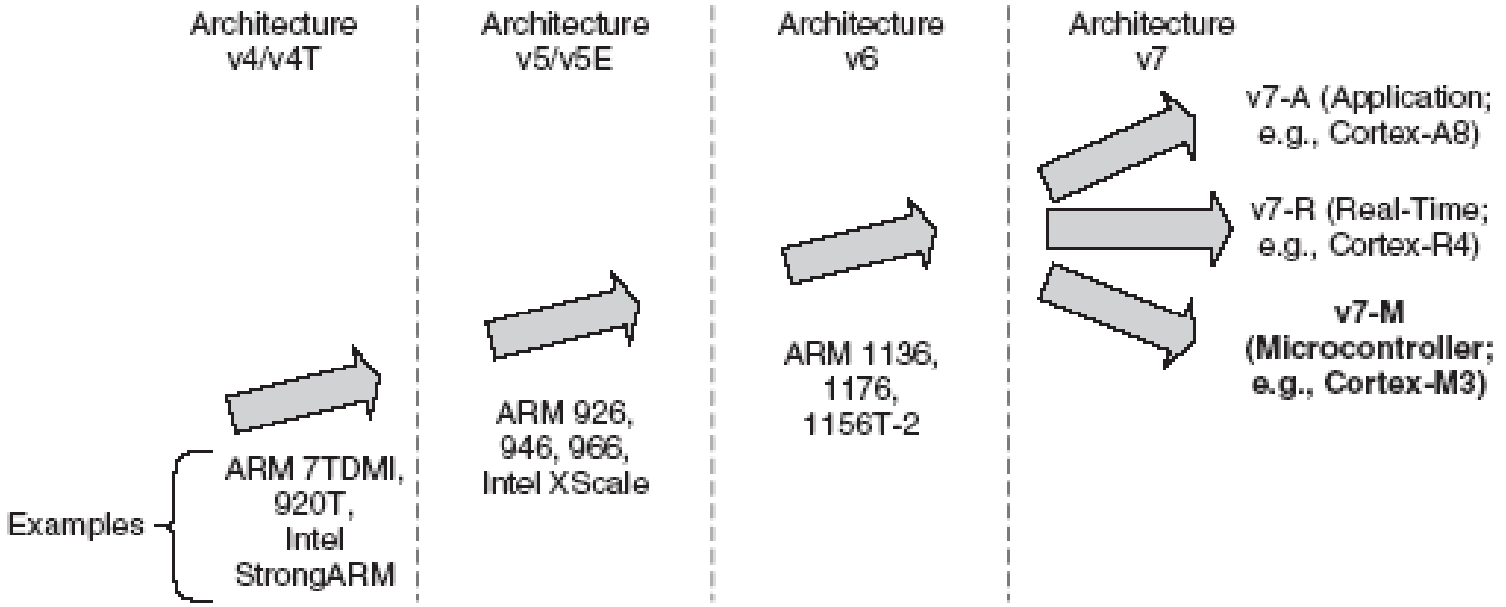
- Found in 1990 as Advanced RISC Machine Ltd. (by apple, Acorn and VLSI technology)
- Ship 2 billion ARM processor each year
- Doesn’t manufacture processor but license the processor designs
- Partner companies create their processors, microcontroller and system-on-chip solution.
- This business model is called intellectual property (IP)

ARM

- Was Acorn Computers, spun off in 1990 and became “Advanced RISC Machine” (ARM)
- Start out to replace 6502 processor
- ARM1, circa 1985
- ARM2, real production, circa 1986
  - 32-bit data bus
  - 26-bit address space (top 6 bits used as status flag)
  - 16 32-bit registers (1 PC)
  - 30,000 transistors
  - No microcode (remember the “R” in the acronym?)
  - No cache
  - Low power
- ARM3, first with 4KB Cache
- ARM6, used in Apple’s Newton
- ARM7TDMI, early most successful ARM core



Evolution of ARM processor Architecture



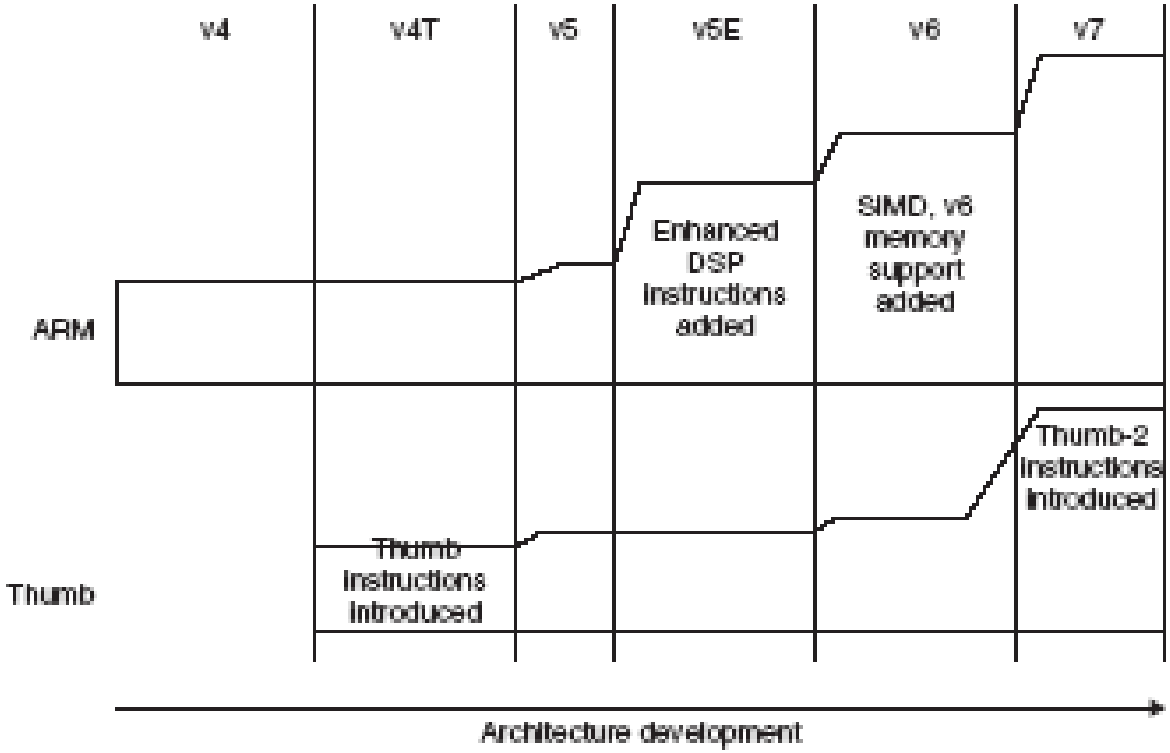
Architecture profile for architecture version 7

- A profile for high performance
- R profile for real-time performance
- M profile for microcontroller-type systems

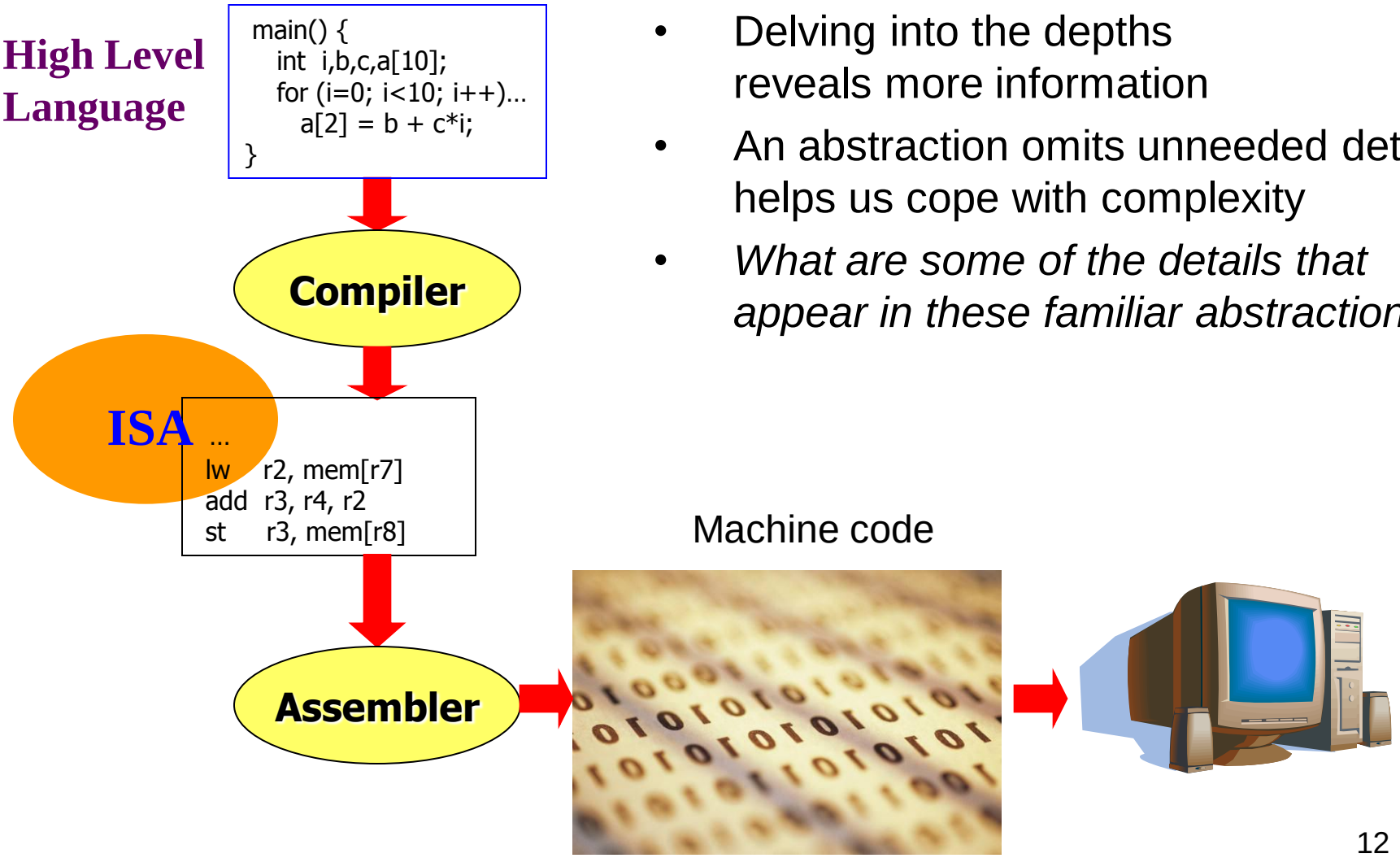
ARM processor name

Processor Name	Architecture Version	Memory Management Features	Other Features
ARM7TDMI	ARMv4T		
ARM7TDMI-S	ARMv4T		
ARM7EJ-S	ARMv5E		DSP, Jazelle
ARM920T	ARMv4T	MMU	
ARM922T	ARMv4T	MMU	
ARM926EJ-S	ARMv5E	MMU	DSP, Jazelle
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E		DSP
ARM968E-S	ARMv5E		DMA, DSP
ARM966HS	ARMv5E	MPU (optional)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU or MPU	DSP, Jazelle
ARM1136J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + multiprocessor cache support	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M3	ARMv7-M	MPU (optional)	NVIC
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + Floating point
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle

Instruction Set Enhancement



Abstraction



- Delving into the depths reveals more information
- An abstraction omits unneeded detail, helps us cope with complexity
- *What are some of the details that appear in these familiar abstractions?*

Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow

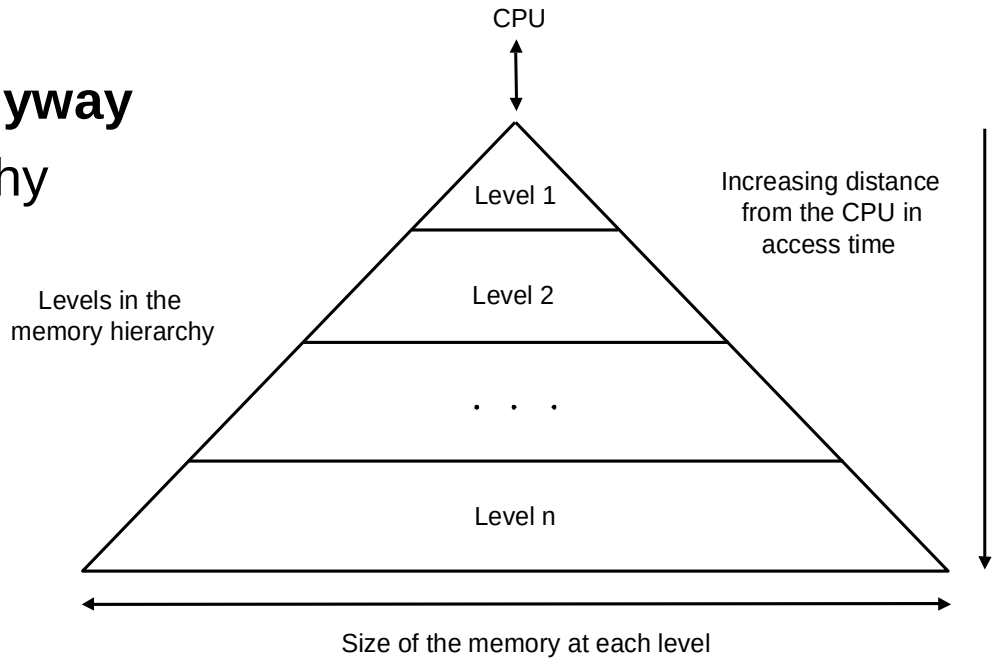
- We'll be working with the ARM instruction set architecture

Design goals: Maximize performance and Minimize cost, Reduce design time

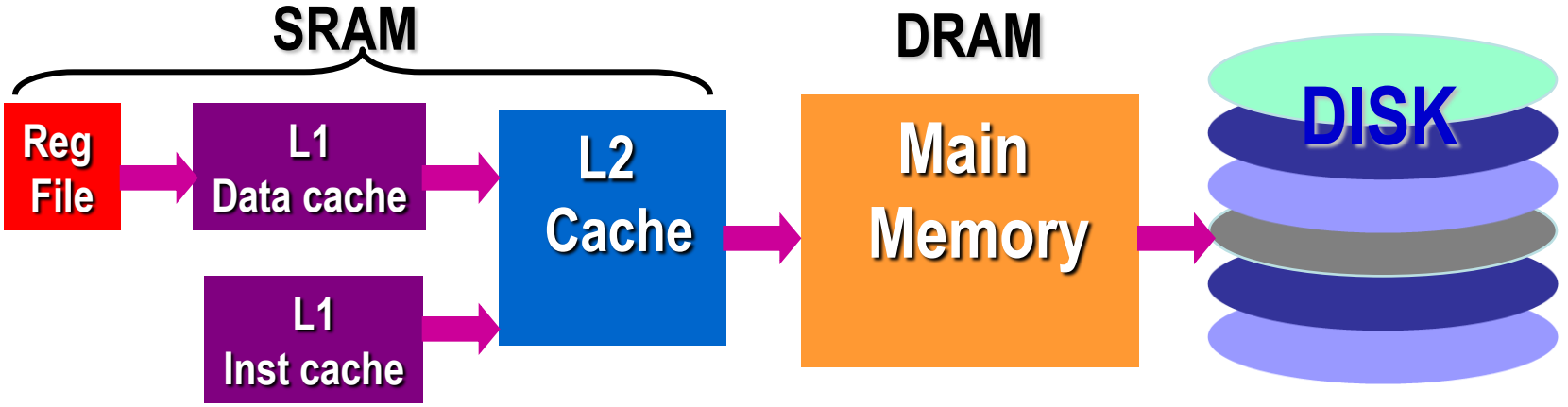
Exploiting Memory Hierarchy

- **Users want large and fast memories! For example:**
  - SRAM access times are 700ps-1ns (1-3 cycles)
  - DRAM access times are 60-100ns (100-250 cycles)
  - Disk access times are ~1 million ns (~3M cycles)

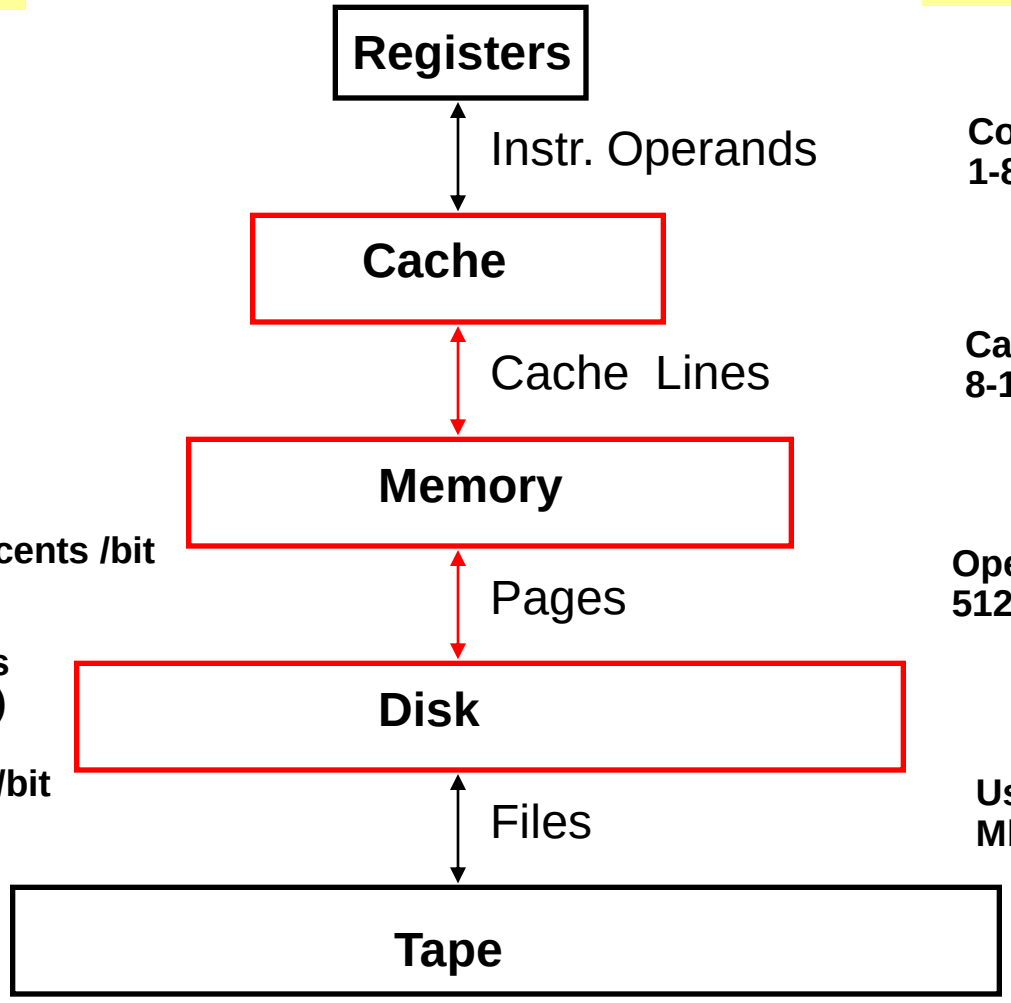
- **Try and give it to them anyway**
  - build a memory hierarchy



Model of Memory Hierarchy



Levels of the Memory Hierarchy



Capacity
Access Time
Cost

CPU Registers
100s Bytes
<10 ns

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns- 500ns
$.0001-.00001 cents /bit

Disk
G Bytes, 10 ms
(10,000,000 ns)
10^{-5} - 10^{-6} cents/bit

Tape
infinite
sec-min
10^{-8}

Staging
Transfer Unit

Compiler
1-8 bytes

Cache controller
8-128 bytes

Operating system
512-4K bytes

User
Mbytes

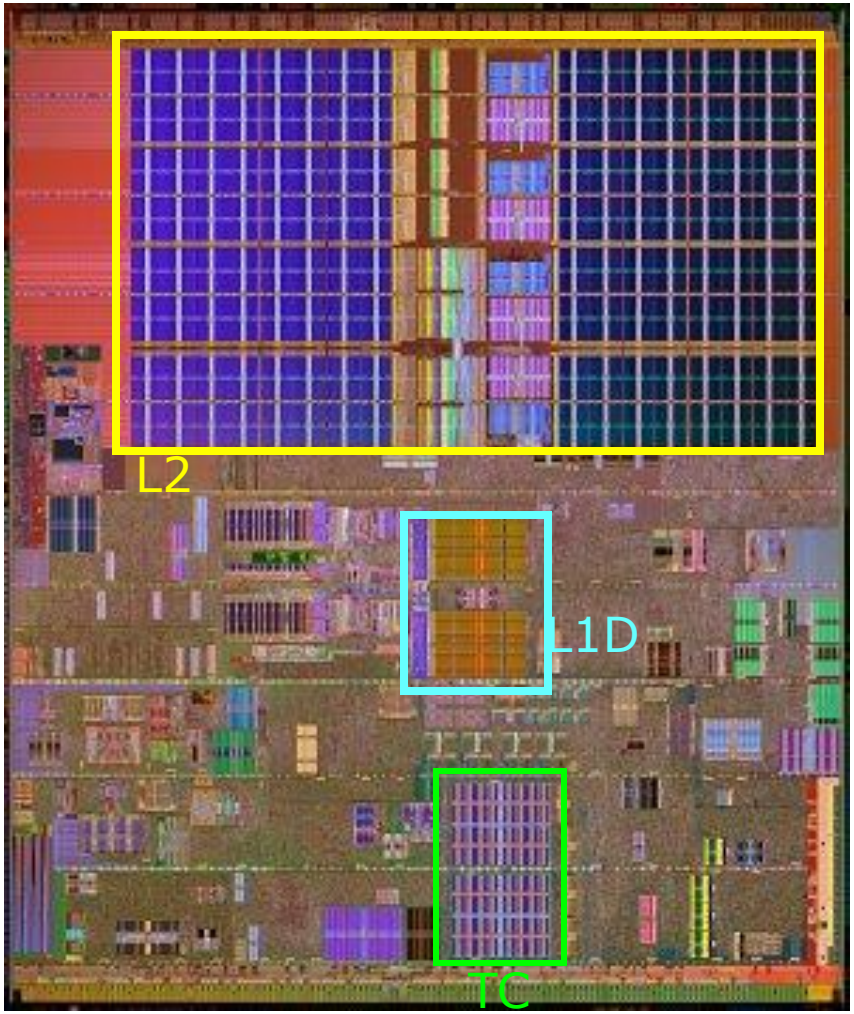
Upper Level

faster

Larger

Lower Level

P4: Prescott w/ 2MB L2 (90nm)

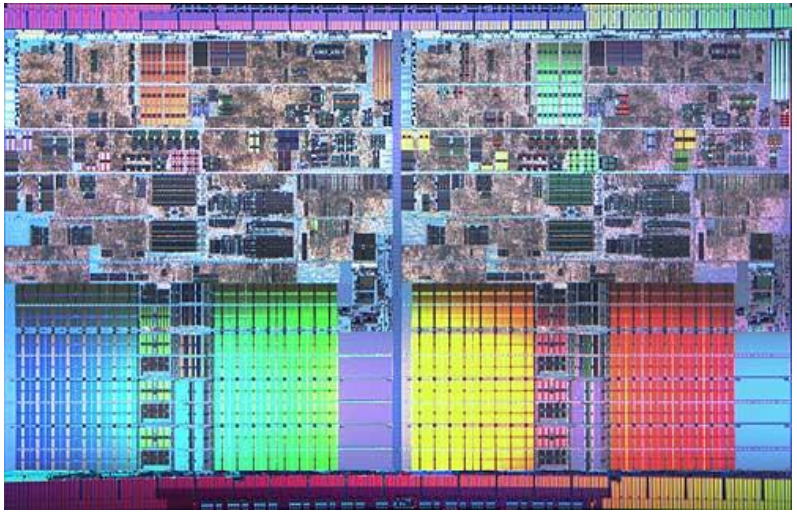


- Prescott runs *very* fast (3.4+ GHz)
- 2MB L2 Unified Cache
- 12K* trace cache (think “I$”)
- 16KB data cache

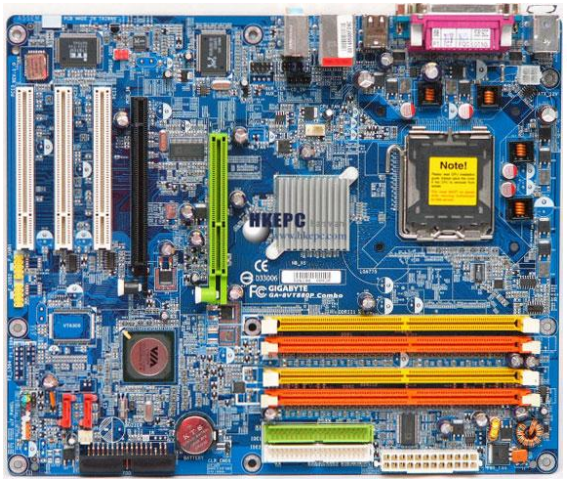
- Where is the cache?
- What about the similar blocks?
- Why the visual differences?
- Why is it square?

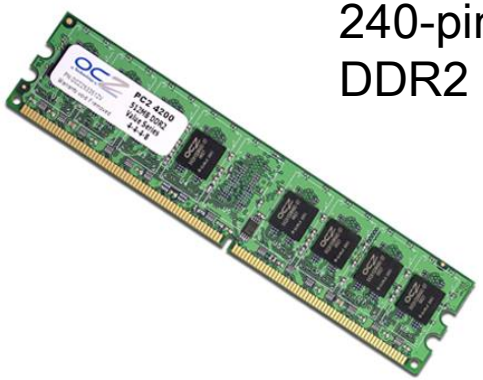
- Check this out:
  - www.chip-architect.com

Memory Photos



Intel Paxville (dual Core) 90nm
8-way 2MB L2 for each core





240-pin
DDR2 DRAM

Register file

- Fast access memory (only 1 cycle)
- Only limited number of register are available
- Processors are commonly 8-bit, 16-bit, 32-bit, or 64-bit, referring to the width of their registers

ARM instruction

- Operand order is fixed (destination first)

  Example:

  C code: `A = B + C`

  ARM code: `add R0, R1, R2`

Format:

Label

  opcode Rd, Rn, Rm …; comments

Rm can be shifted

ARM arithmetic

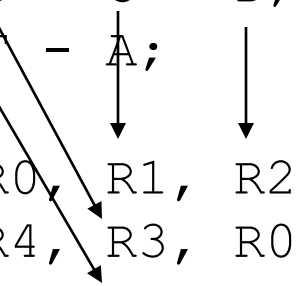
- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code:

```
A = B + C + D;
E = F - A;
```

ARM code:

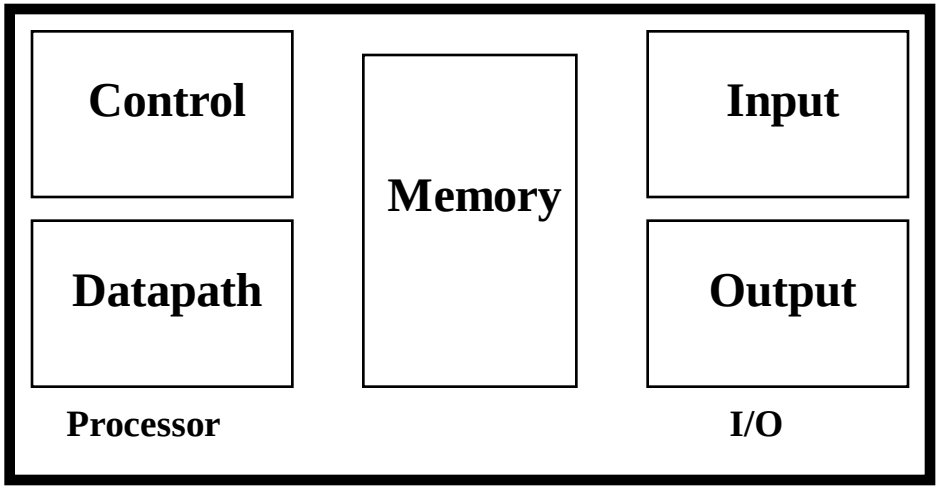
```
add R0, R1, R2
add R4, R3, R0
sub R6, R5, R4
```



- Operands must be registers, 16 registers provided
- All memory accesses are accomplished via loads and stores
  - A common feature of RISC processors

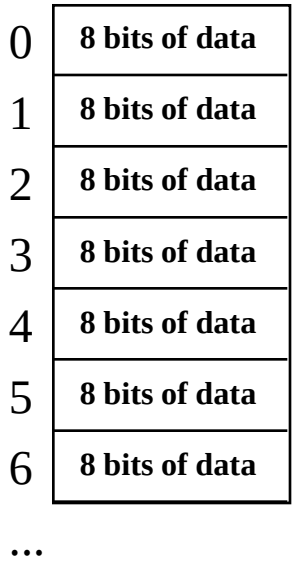
Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 16 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



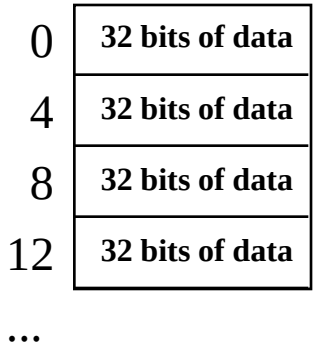
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words“
- ARM provides LDR/LDRH/LDRB and STR/STRH/STRHB instructions
- For ARM, a word is 32 bits or 4 bytes.
  - (Intel’s word=16 bits and double word or dword=32bits)



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned

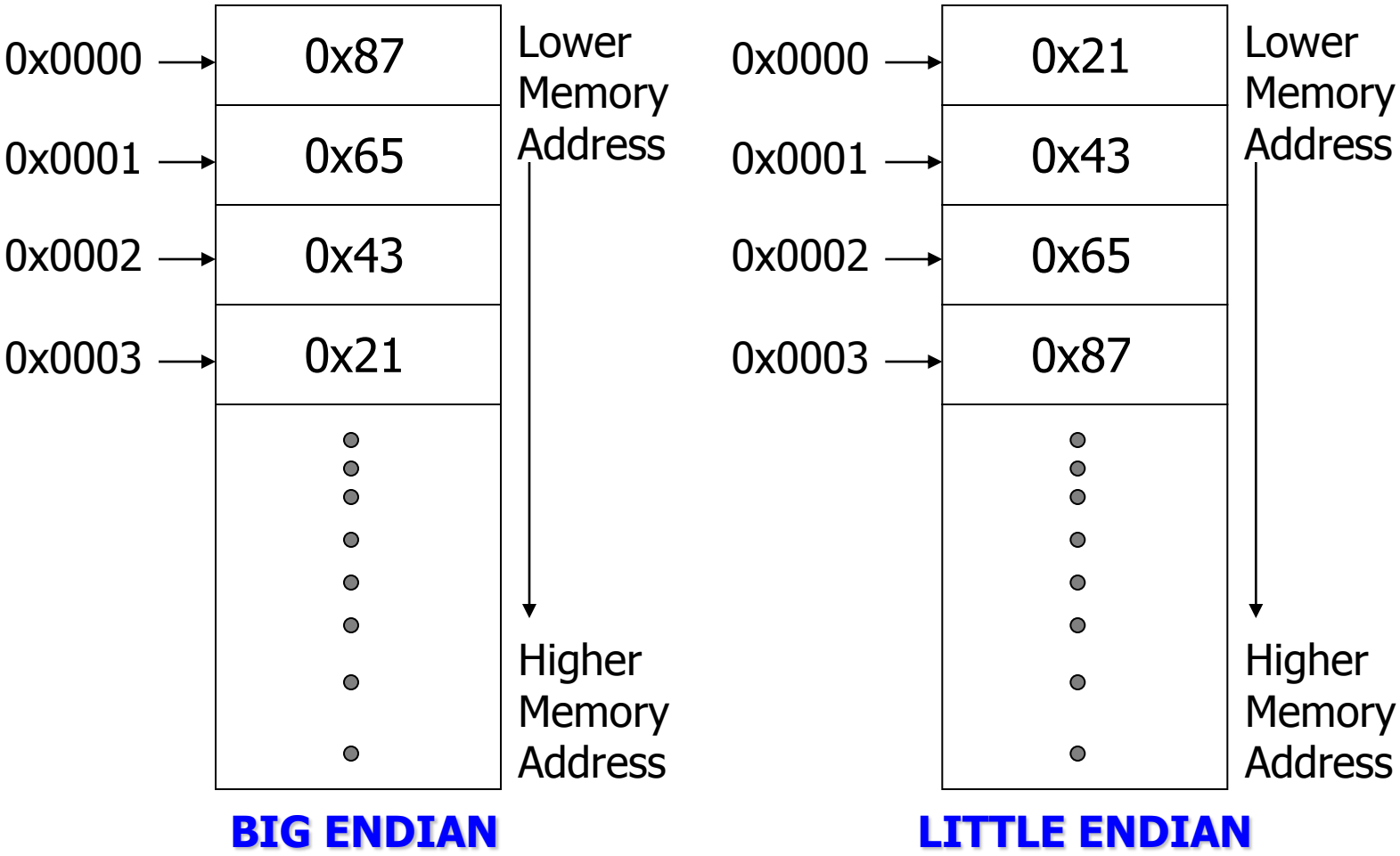
  i.e., what are the least 2 significant bits of a word address?

Endianness [defined by Danny Cohen 1981]

- Byte ordering — How a multiple byte data word stored in memory
- Endianness (from Gulliver's Travels)
  - Big Endian
    - Most significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Sun Sparc, PowerPC
  - Little Endian
    - Least significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Intel x86
- Some embedded & DSP processors would support both for interoperability

Example of Endian

- Store 0x87654321 at address 0x0000, byte-addressable

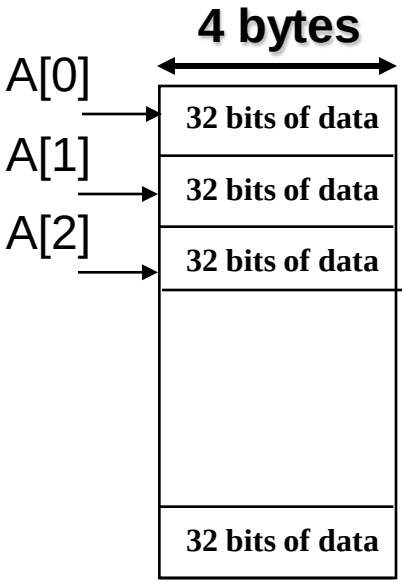


Register and Memory Access

- In the ARM architecture
  - Memory is byte addressable
  - 32-bit addresses
  - 32-bit processor registers
- Word addresses must be aligned, i.e., they must be multiple of 4
  - Both little-endian and big-endian memory addressing are supported
- When a byte is loaded from memory into a processor register of stored from a register into the memory
  - Support both little endian and big endian

Instructions

- Load (LDR) and store (STR) instructions
- Example:



C code:

```
long A[100];
A[9] = h + A[8];
```

ARM code:

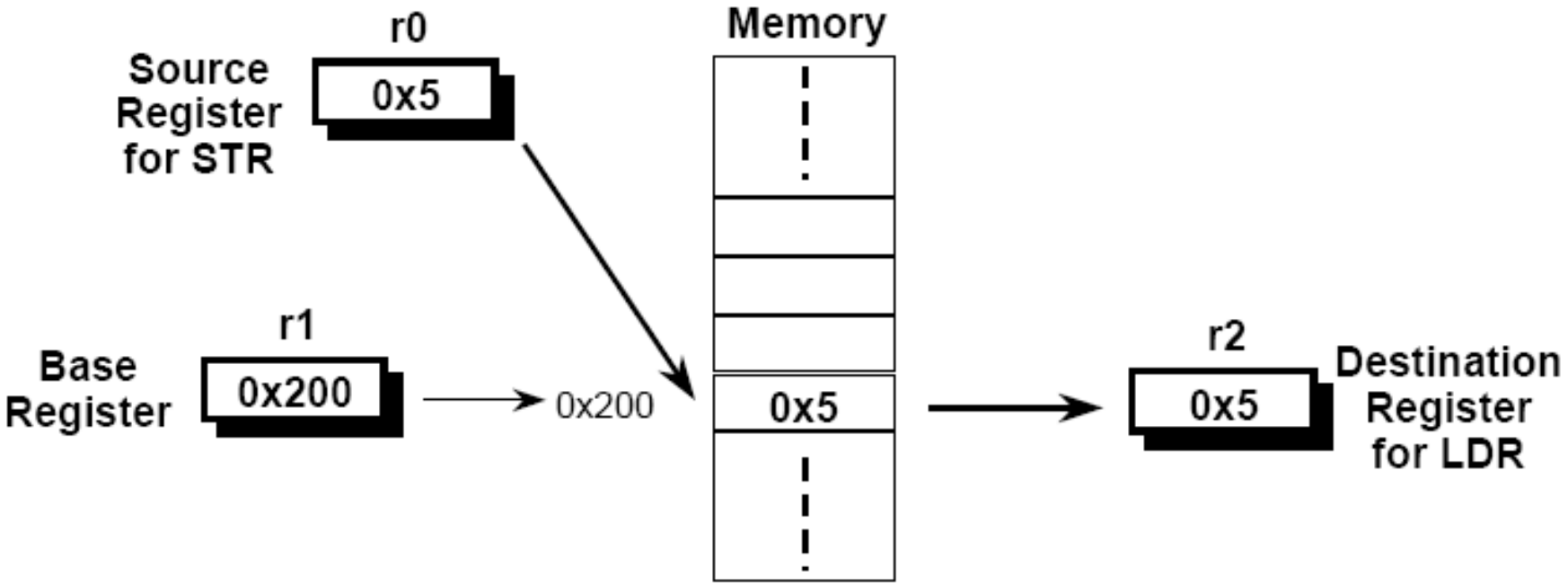
```
LDR  R0, [R3,32]
ADD R0, R2, R0
STR  R0, [R3,36]
```

- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Load/Store Word

STR R0, [R1]

LDR R2,[R1]



Our First Example

```
swap(int v[], int k);
{
int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
        mul  r2, r5, 4
        add  r2, r4, r2
        ldr   r5, [r2,0]
        ldr   r6, [r2,4]
        str   r6, [r2,0]
        str   r5, [r2,4]
        mov  PC, r14
```

So far we've learned:

- ARM
  - — loading words but addressing bytes
  - — arithmetic on registers only

- Instruction / Meaning

```
add r1, r2, r3          r1 = r2 + r3
sub r1, r2, r3          r1 = r2 - r3
ldr r1, [r2,100]        r1 =Memory[r2+100]
str r1, [r2,100]        Memory[r2+100]=r1
```

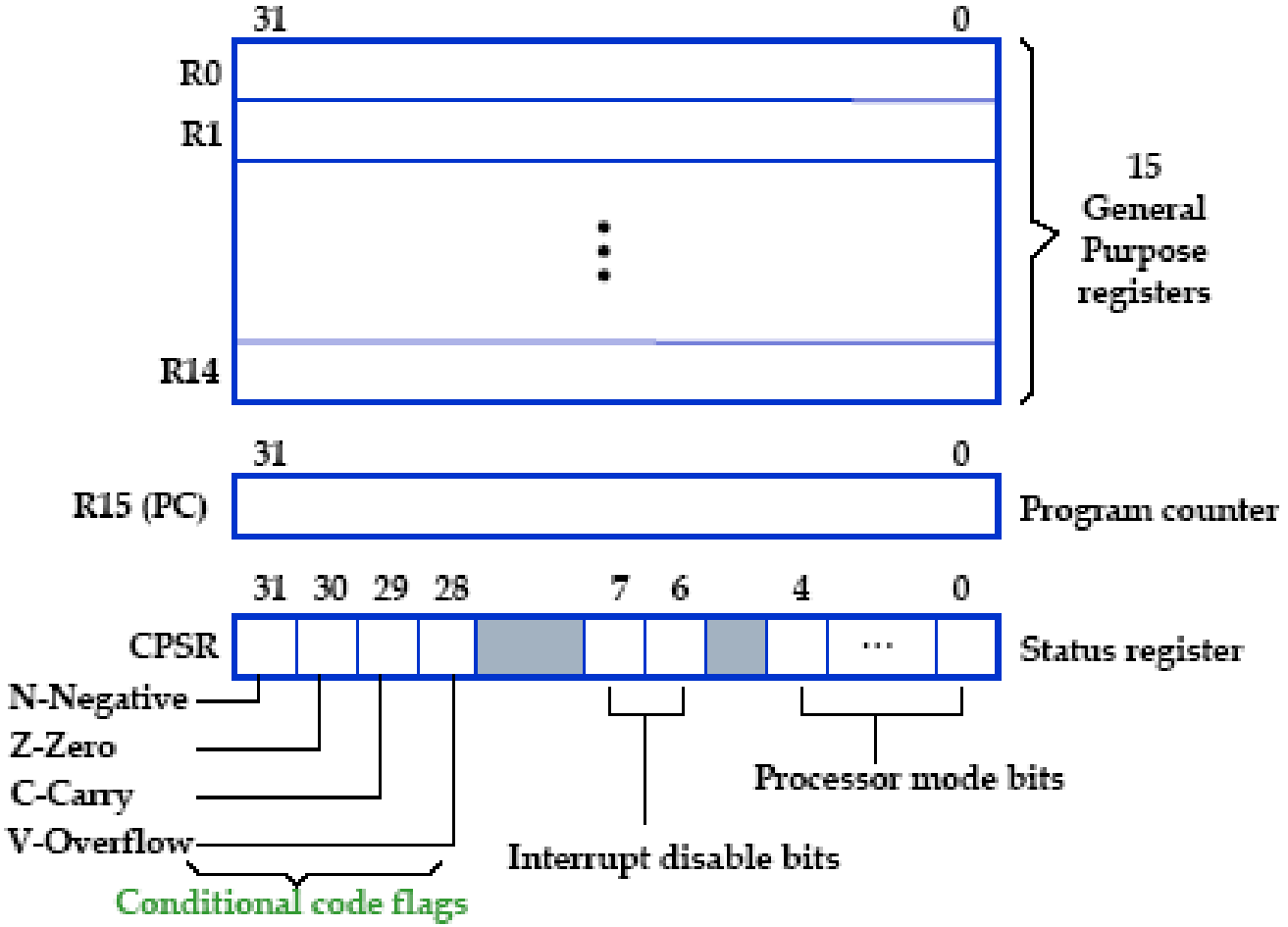
Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add r0, r1, r2`
  - registers have values, r`0=9, r1=17, r2=18`

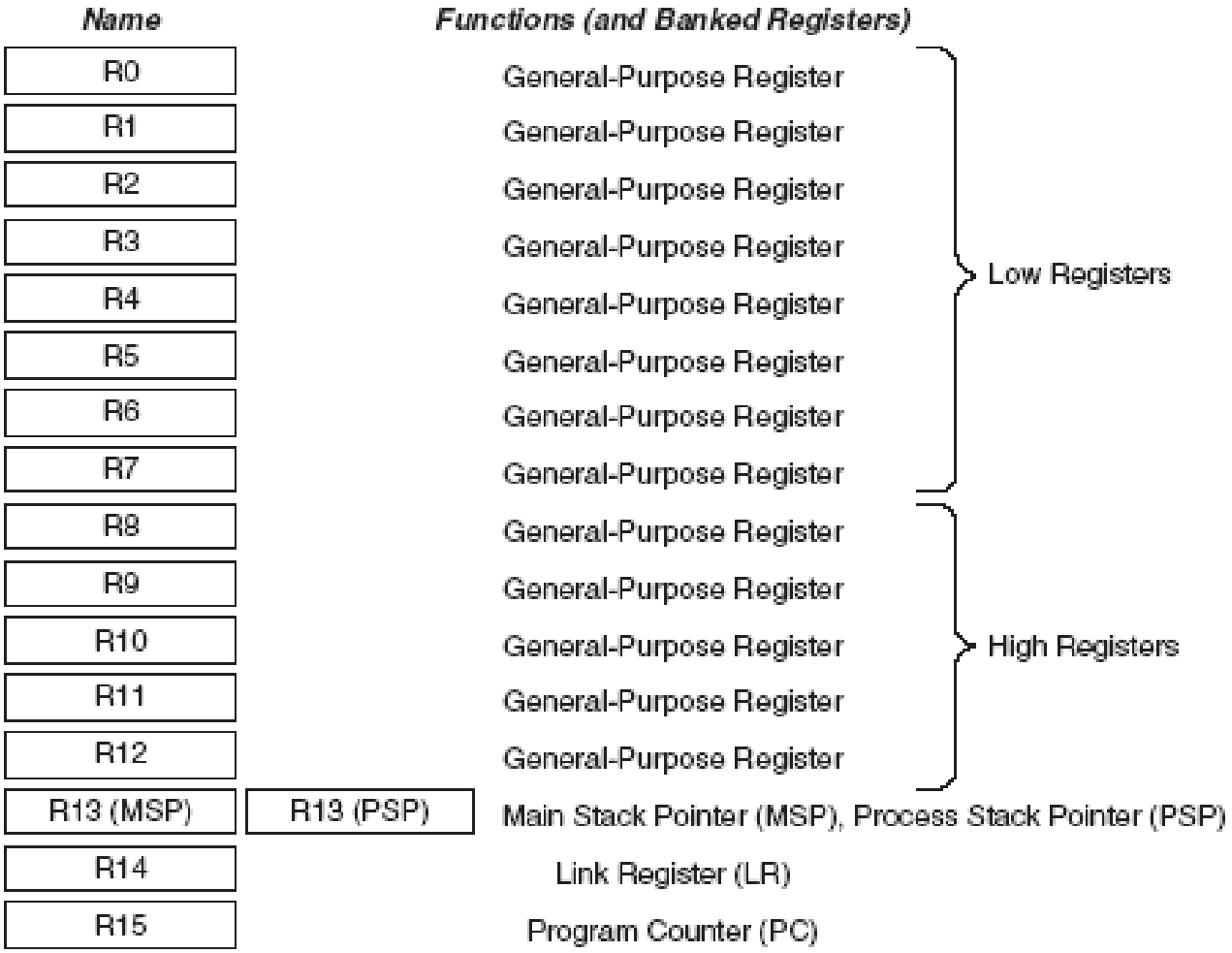
- Instruction Format:

cond	001	OPCODE	rn	rd	rs	0	shift	1	rm
0000	001	01001	0001	0000	0000	0	00	1	0010

Register Structure



Registers in ARM



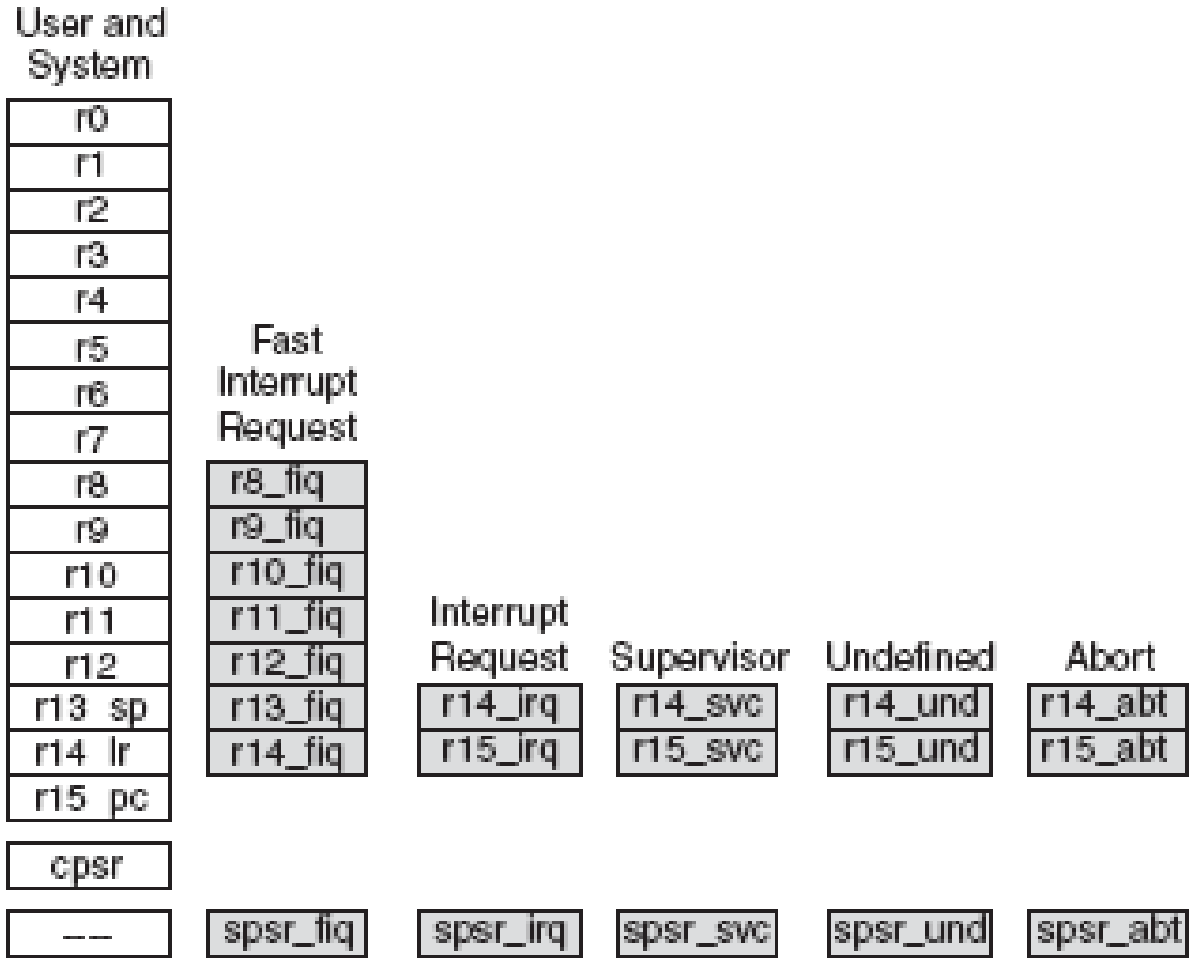
Register structure

- There are 15 additional general-purpose registers called the banked registers
  - They are duplicates of some of the R0 to R14 registers
  - They are used when the processor switches into supervisor or interrupt modes
- Saved copies of the status register are also available in the supervisor or interrupt modes

Processor mode

Mode	Abbreviation	Privileged	Mode bits[4:0]
Abort	*abt*	*yes*	*1 0 1 1 1*
Fast Interrupt Request	*fiq*	*yes*	*1 0 0 0 1*
Interrupt Request	*irq*	*yes*	*1 0 0 1 0*
Supervisor	*svc*	*yes*	*1 0 0 1 1*
System	*sys*	*yes*	*1 1 1 1 1*
Undefined	*und*	*yes*	*1 1 0 1 1*
User	*usr*	*no*	*1 0 0 0 0*

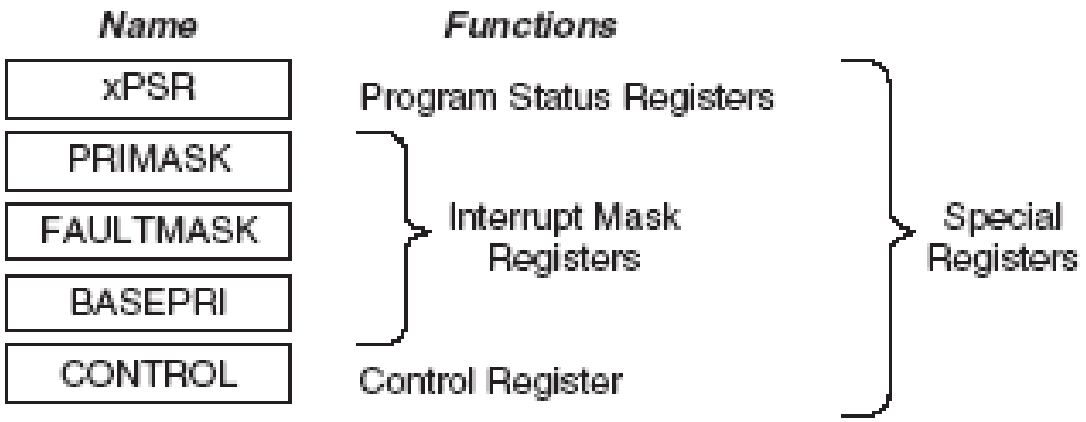
Register file structure



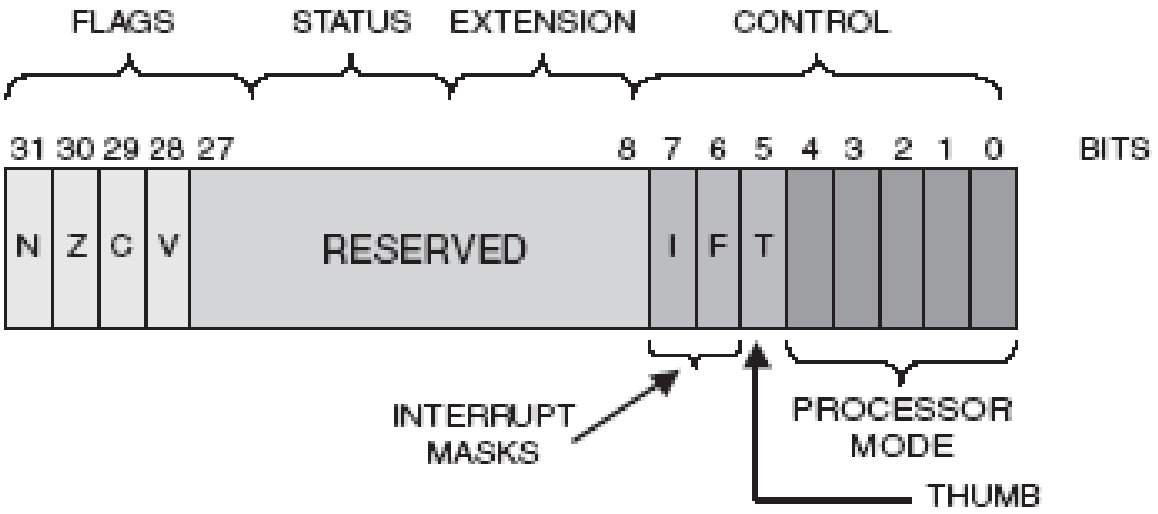
Special registers

- Program Status Registers (PSRs)
- Interrupt Mask Registers
- Control Registers (Control)

Special Registers



Status Register

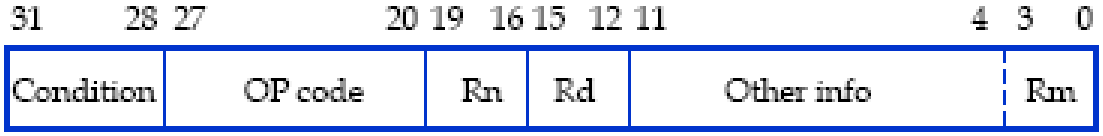


Questions?

- What registers are used to store the program counter and link register?
- What is r13 often used to store?

ARM Instruction Format

- Each instruction is encoded into a 32-bit word
- Access to memory is provided only by Load and Store instructions
- The basic encoding format for instructions such as Arithmetic, and Logic function is shown below



- An instruction specifies a conditional execution code (Condition), the OP code, two or three registers (Rn, Rd, and Rm) and some other information

Conditional Execution of Instructions

- A distinctive and somewhat unusual feature of ARM processors is that all instructions are conditionally executed
  - Depending on a condition specified in the instruction
- The instruction is executed only if the current state of the processor condition code flag satisfies the condition specifies in bit b31-b28 of the instruction
  - Thus the instructions whose condition is not meet the processor condition code flag are not executed
- One of the conditions is used to indicate that the instruction is always executed

Example of conditional execution

SUB R3, R10, R5

SUBNE R3, R10, R5

SUBS R3, R10, R5

SUBNES R3, R10, R5

Conditional execution

Code	Description	Flags	OP-Code[31:28]
EQ	Equal to zero	$Z = 1$	0 0 0 0
NE	Not equal to zero	$Z = 0$	0 0 0 1
CS HS	Carry set / unsigned higher or the same	$C = 1$	0 0 1 0
CC LO	Carry cleared / unsigned lower	$C = 0$	0 0 1 1
MI	Negative or minus	$N = 1$	0 1 0 0
PL	Positive or plus	$N = 0$	0 1 0 1
VS	Overflow set	$V = 1$	0 1 1 0
VC	Overflow cleared	$V = 0$	0 1 1 1
HI	unsigned higher	$Z * \overline{C}$	1 0 0 0
LS	unsigned lower or the same	$Z + \overline{C}$	1 0 0 1
GE	signed greater than or equal	$(N*V) + (\overline{N}*\overline{V})$	1 0 1 0
LT	signed less than	N xor V	1 0 1 1
GT	signed greater than	$(N*\overline{Z}*V) + (\overline{N}*\overline{Z}*\overline{V})$	1 1 0 0
LE	signed less than or equal	$Z + (N \text{ xor } V)$	1 1 0 1
AL	always (unconditional)	not used	1 1 1 0
NV	never (unconditional)	not used	1 1 1 1

Modifier

- S modifier: update condition flags (cmp instruction doesn’t require this flag)
- R modifier: round to the nearest, otherwise truncate

Arithmetic Instructions

- The basic expression for arithmetic instructions is

  OPcode Rd, Rn, Rm
- For example, ADD R0, R2, R4

  Performs the operation R0 = R2+R4
- SUB R0, R6, R5

  Performs the operation R0 = R6-R5
- Immediate mode: ADD R0, R3, #17

  Performs the operation R0 = R3+17
- The second operand can be shifted or rotated before being used in the operation

  For example, ADD R0, R1, R5, LSL #4 operates as follows: the second operand stored in R5 is shifted left 4-bit positions (equivalent to [R5]x16), and its is then added to the contents of R1; the sum is placed in R0

Logic Instruction

- The logic operations AND, OR, XOR, and Bit-Clear are implemented by instructions with the OP codes AND, ORR, EOR, and BIC.

  For example

  AND R0, R0, R1: performs R0 = R0 & R1

- The Bit-Clear instruction (BIC) is closely related to the AND instruction.

  It complements each bit in operand Rm before ANDing them with the bits in register Rn.

  For example, BIC R0, R0, R1. Let R0=02FA62CA, R1=0000FFFF. Then the instruction results in the pattern 02FA0000 being placed in R0

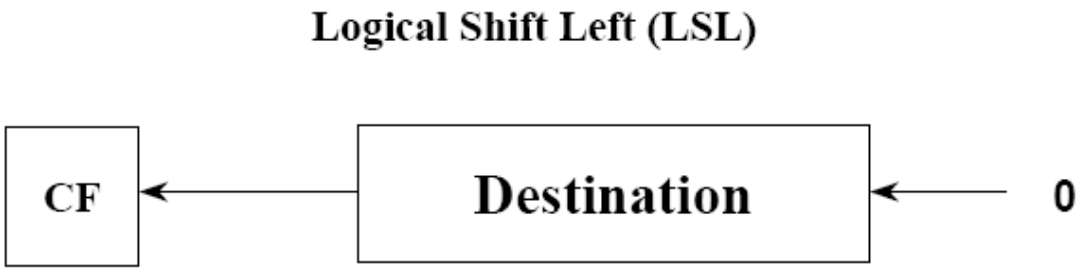
- The Move Negative instruction complements the bits of the source operand and places the result in Rd. For example, MVN R0, R3

Barrel shifter

- There is no shift instruction in ARM
- Instead, it provides the barrel shifter to carry out shift operations as part of other instructions

Barrel shifter –left shift

- Shift left by specific amount
  e.g. LSL #5 = multiply by 32

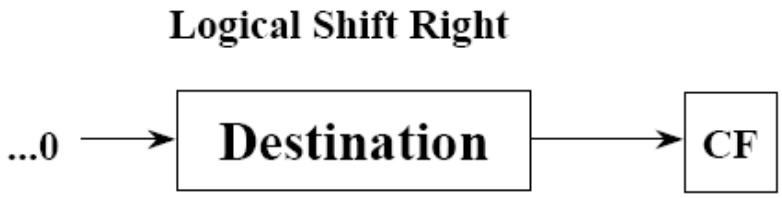


Barrel shifter –right shift

- Logical shift right

  e.g. LSR # 5 =

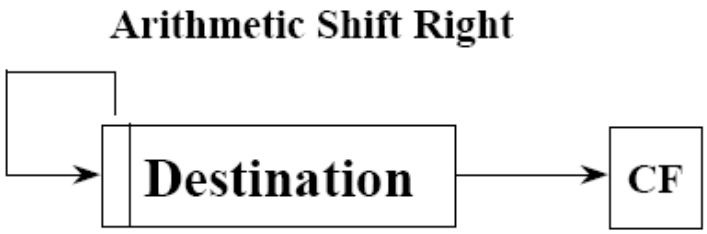
  divided by 32



- Arithmetic shift right

  e.g. ASR #5 =

  divided by 32 and preserve the sign bit

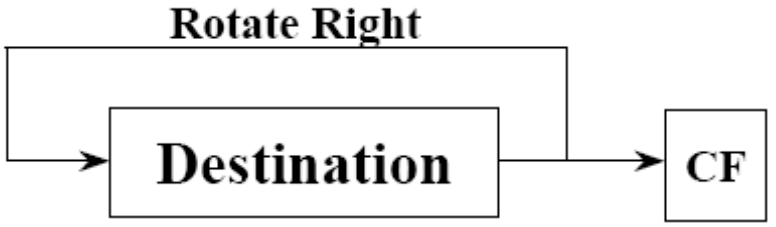


Barrel shifter –rotate right

- Rotate right

similar to shifter but wrap around LSB bit

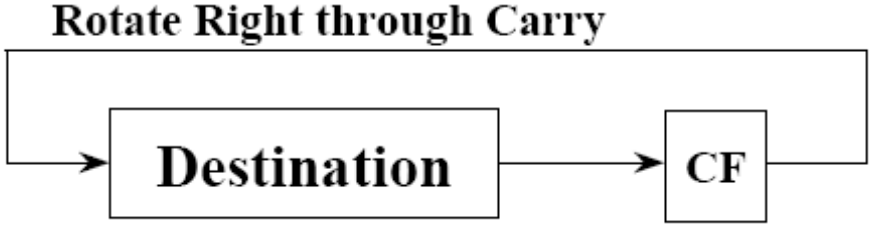
ROR #5



- Rotate right extended

Use C flag as 33 bits

RRX #5



Barrel shifter

If R1 = 0x 0012, what is the value of R6 and R9?

MOV R6, R1

MOV R6, R1, LSL #3

MOV R9, #3

MOV R6, R1, LSL R9

Barrel shift operation

Mnemonic	Operation	Shift Amount
LSL	*Logical Shift Left*	*#0-31, or register*
LSR	*Logical Shift Right*	*#1-32, or register*
ASR	*Arithmetic Shift Right*	*#1-32, or register*
ROR	*Rotate Right*	*#1-32, or register*
RRX	*Rotate Right Extended*	*33*

Example

- What is the meaning of below instruction?

  ADD r3, r2, r1, LSL #3;

If r1 = 0x0010

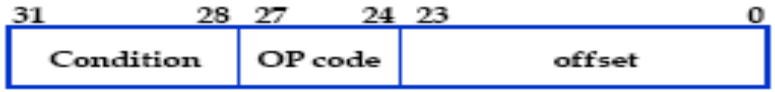
r2 = 0x0001

r3 = 0x0100

Branch Instruction

- Conditional branch instructions contain a signed 24-bit offset that is added to the updated contents of the Program Counter to generate the branch target address

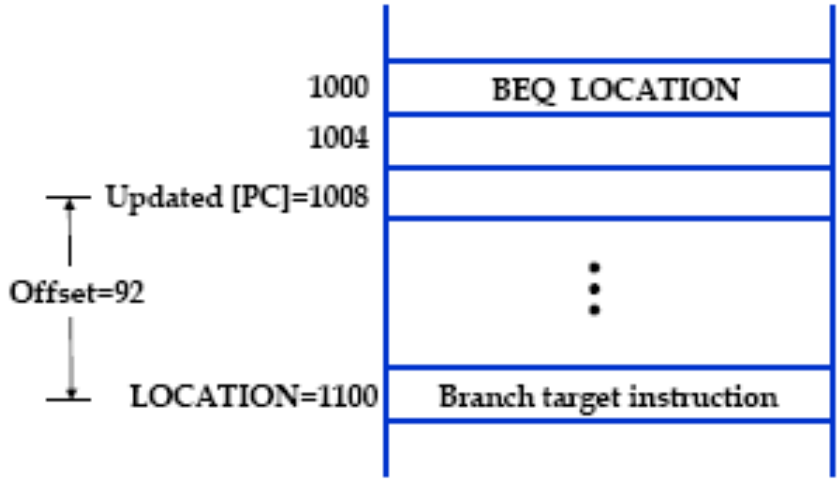
- The format for the branch instructions is shown as below



- Offset is a signed 24-bit number. It is shifted left two-bit positions (all branch targets are aligned word addresses), signed extended to 32 bits, and added to the updated PC to generate the branch target address
- The updated points to the instruction that is two words (8 bytes) forward from the branch instruction

ARM Branch Instructions

- The BEQ instruction (Branch if Equal to 0) causes a branch if the Z flag is set to 1
- Branch relative address is with current PC + 8



Setting Condition Codes

- Some instructions, such as Compare, given by

  CMP Rn, Rm which performs the operation Rn-Rm have the sole purpose of setting the condition code flags based on the result of the subtraction operation
- The arithmetic and logic instructions affect the condition code flags only if explicitly specified to do so by a bit in the OP-code field. This is indicated by appending the suffix S to the OP-code

  For example, the instruction ADDS R0, R1, R2 set the condition code flags

  But ADD R0, R1, R2 does not

Assembly Language

- An EQU directive can be used to define symbolic names for constants
- For example, the statement

  TEN EQU 10
- When a number of registers are used in a program, it is convenient to use symbolic names for them that relate to their usage

  The RN directive is used for this purpose

  For example, COUNTER RN 3 establishes the name COUNTER for register R3
- The register names R0 to R15, PC (for R15), and LR(for R14) are predefined by the assembler

  R14 is used for a link register (LR)

Questions?

- Specify ARM assembly instruction to do the following:
  - R0 = 16
  - R1 = R0 *4
  - R0 = R1 /16 (maintain the sign bit)
  - R1 = R2*5

Questions?

Assume that r0 = 32, r1 = 1

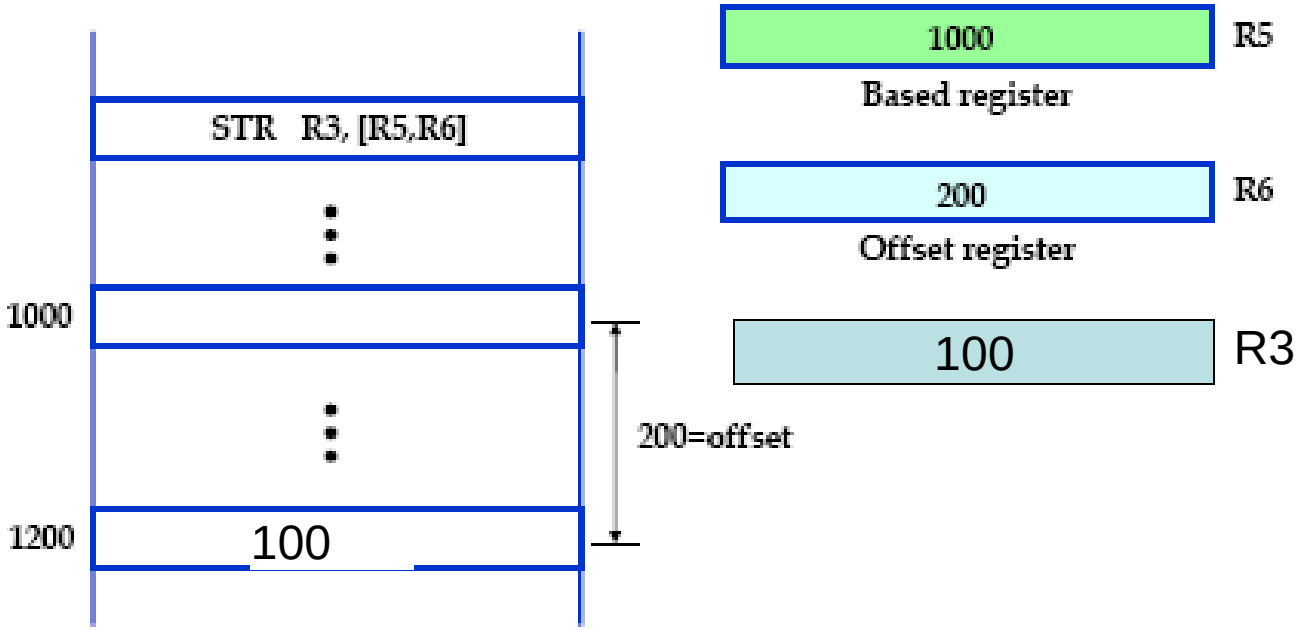
- What will the following instructions do?
  - ADDS r0, r1, r1, LSL #2
  - SUB r0, r0, r1, LSL #4

Memory Addressing Modes

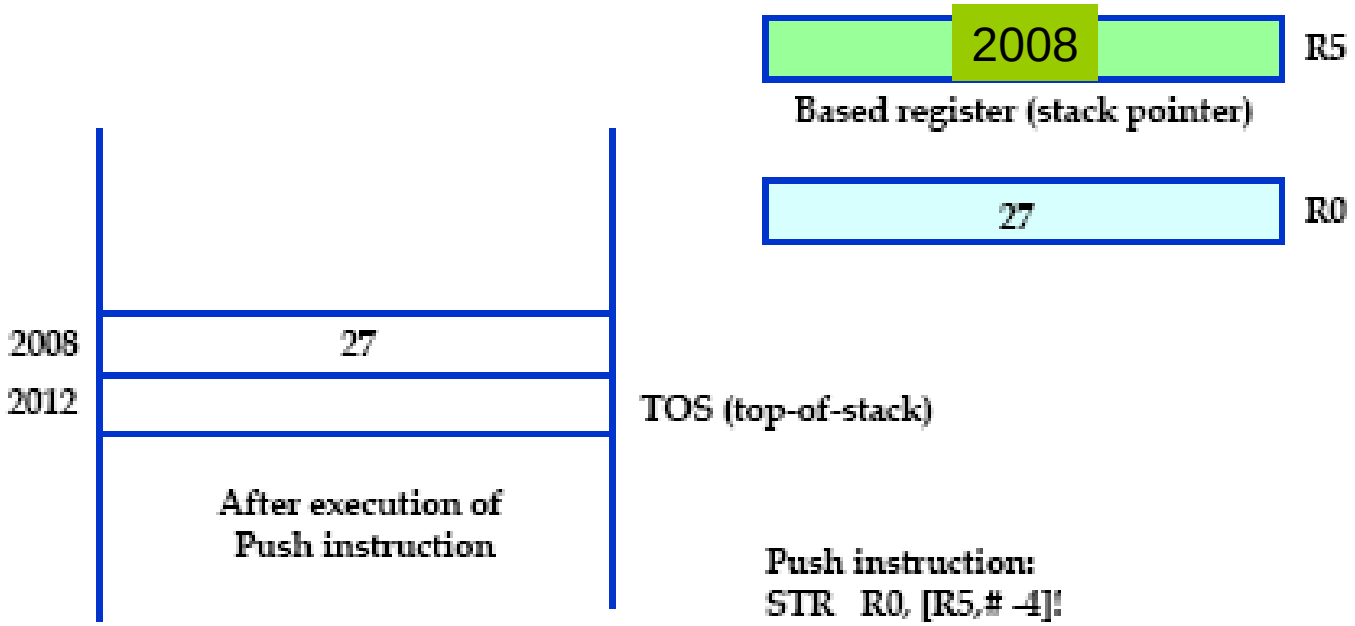
- Pre-indexed mode
  - The effective address of the operand is the sum of the contents of the based register Rn and an offset value
- Pre-indexed with writeback mode (auto-indexing)
  - The effective address of the operand is generated in the same way as in the Pre-indexed mode, and then the effective address is written back into Rn
- Post-indexed mode
  - The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn

Name	Assembly	Addressing function
With immediate offset:		
Pre-indexed	[Rn,#offset]	EA=mem [Rn+offset]
Pre-indexed with writeback	[Rn,#offset]!	EA=mem [Rn+offset]; Rn = Rn + offset;
Post-indexed	[Rn],#offset	EA= mem [Rn]; Rn = Rn + offset;
With offset and Rm		
Pre-indexed	[Rn,±Rm,#offset]	EA= mem[Rn ± Rm+offset];
Pre-indexed with writeback	[Rn,±Rm,#offset]!	EA= mem[Rn ± Rm+offset]; Rn = Rn ± Rm + offset
Post-indexed	[Rn],±Rm,#offset	EA= mem[Rn]; Rn = Rn ± Rm + offset
Relative	Location	EA = Location = [PC] + offset

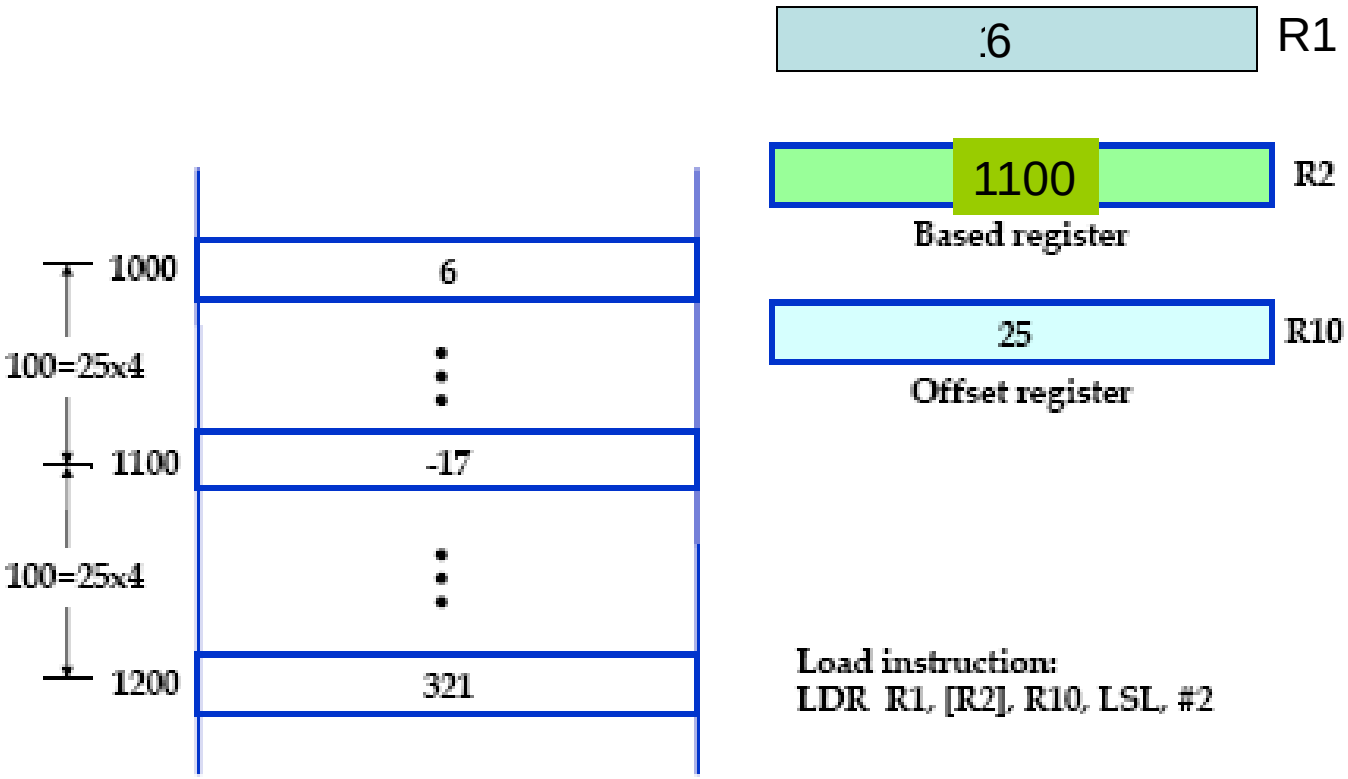
Pre-Indexed Addressing Mode



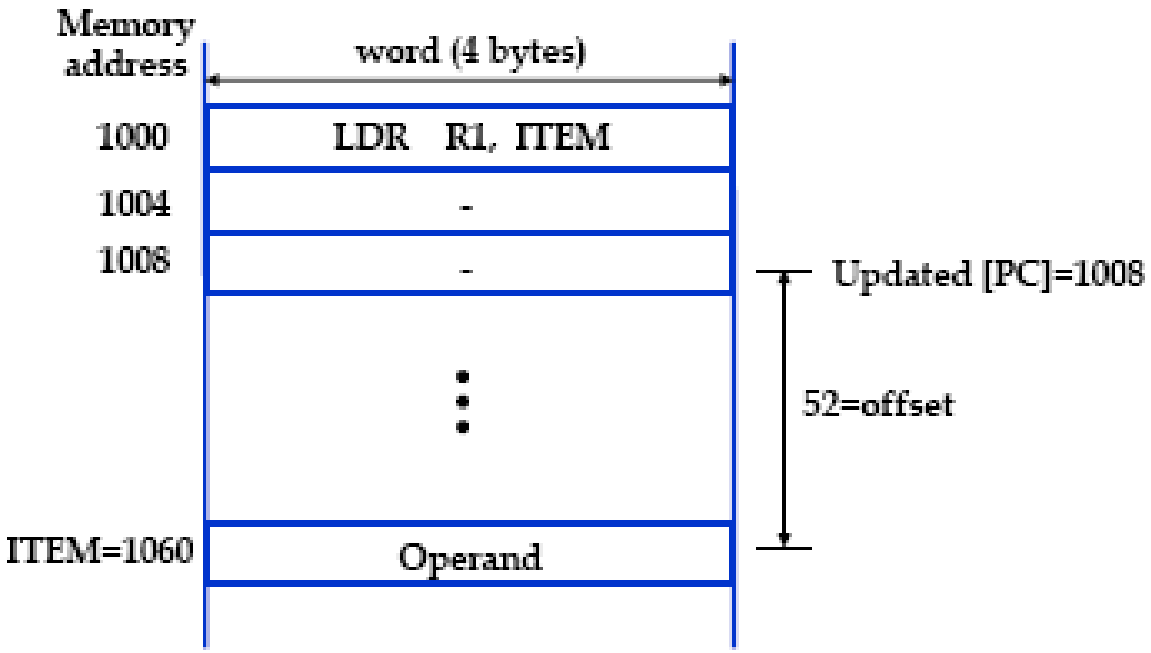
Pre-Indexed Addressing with Writeback



Post-Indexed Addressing



Relative Addressing Mode



The operand must be within the range of 4095 bytes forward or backward from the updated PC

Memory instructions

Mnemonic	Description	Operation
LDR	*Load a word from memory into a register*	*Register < mem32*
STR	*Store the word contents of a register in memory*	*mem32 < Register*
LDRB	*Load a byte from memory into a register*	*Register < mem8*
STRB	*Store the byte contents of a register in memory*	*mem8 < Register*
LDRH	*Load a half-word from memory into a register*	*Register < mem16*
STRH	*Store the half-word contents of a register in memory*	*mem16 < Register*
LDRSB	*Load a signed byte into a register*	*Register < Sign extended mem8*
LDRSH	*Load a signed half-word into a register*	*Register < Sign extended mem16*

Addressing mode

Index mode	Description	Register offset	Immediate offset	Scaled register
Preindex without write back	*Address calculated prior to being used*	`[Rn, ±Rm]`	`[Rn, #±Imm12]`	`[Rn, ±Rm, shift #imm]`
Preindex with write back	*Address calculated prior to being used; base <- base + offset*	`[Rn, ±Rm]!`	`[Rn, #±Imm12]!`	`[Rn, ±Rm, shift #imm]!`
Postindex with write back	*Address calculated after being used; base <- base + offset*	`[Rn], ±Rm`	`[Rn], #±Imm12`	`[Rn], #±Rm, shift #imm`

Example

```
LDR  r12, [r0,+r7]
LDR  r12, [r0,-#0x6A0]!
STR  r12, [r0], +#0x6A0
STRH r12,[r0,+r7]!
LDR  r12, [r0, -r7, LSL #3]
LDREQ r12, [r0, -r7, LSL #3]
```

Load/Store Multiple operands

- In ARM processor, there are two instructions for loading and storing multiple operand
  - They are called Block transfer instructions
- Only word operands are allowed, and the OP codes used are LDM and STM
- The memory operands must be in successive word locations
- All of the forms of pre- and post-indexing with and without writeback are available
- They operate on a Base register Rn specified in the instruction and offset is always 4

Multiple load/store

LDM : load multiple registers

SDM : store multiple registers

E.g. :

LDMIA R10, {r0-r3}

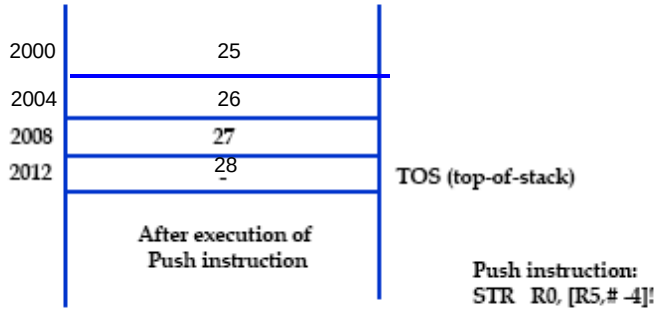
LDMIA R10!, {r0-r3}

Example

- LDMIA R10, {r0-r1}

R10 = 2004

- LDMDB R10, {r0-r1}



Operations

Mnemonic	Description	Comments
IA	*Increment After*	*First data transfer occurs at memory location pointed to in base register, Rn. Subsequent transfers are from successively higher memory locations.*
IB	*Increment Before*	*First data transfer occurs at memory location 4 bytes higher in memory than initial value in base register, Rn. Subsequent transfers are from successively higher memory locations.*
DA	*Decrement After*	*First data transfer occurs at memory location pointed to in base register, Rn. Subsequent transfers are from successively lower memory locations.*
DB	*Decrement Before*	*First data transfer occurs at memory location 4 bytes lower in memory than initial value in base register, Rn. Subsequent transfers are from successively lower memory locations.*

Stack operations

- Full ascending (FA) is equivalent to IB mode
- Full Descending (FD) is equivalent to DB mode
- Empty ascending (EA) is equivalent to IA mode
- Empty Descending (ED) is equivalent to DA mode

Example of stack instruction

STMFD SP!, {r7-r10}

An Example of Adding Numbers (sum from 0 to N-1)

```
        MOV R1, #N          Move count into R1
        LDR R2, POINTER     Load address NUM1 into R2
        MOV R0, #0          Clear accumulator R0
Loop    LDR R3, [R2], #4    Load next number into R3
        ADD R0, R0, R3      Add number into R0
        SUBS R1, R1, #1     Decrement loop counter R1
        BGT LOOP            Branch back if not done
        STR R0, SUM         Store sum
```

Assuming that the memory location POINTER, and SUM are within the range reachable by the offset relative to the PC

GT: signed greater than

BGT: Branch if Z=0 and N=0

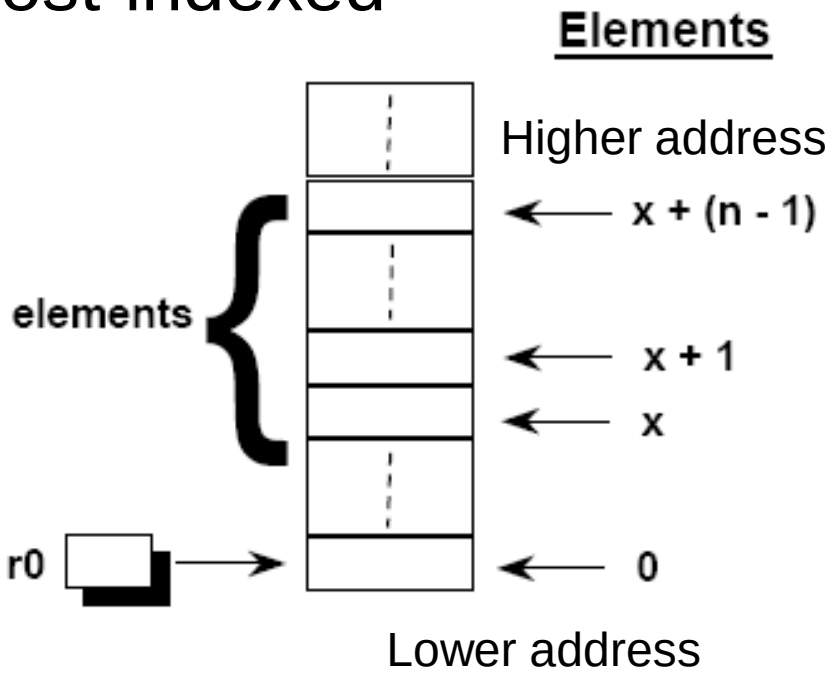
Questions?

- Write a segment of ARM code that add together element x till element x+(n-1)
- The segment should use post-indexed
- Each element is word size
- Assume that

r0 point to the start of array

r1 = x

r2 = n



Sample solution

```
__ADD r0, r0, r1, LSL#2        ; Set r0 to address of element x
  ADD r2, r0, r2, LSL#2        ; Set r2 to address of element n+1
  MOV r1, #0                   ; Initialise counter
loop
  LDR r3, [r0], #4             ; Access element and move to next
  ADD r1, r1, r3               ; Add contents to counter
  CMP r0, r2                   ; Have we reached element x+n?
  BLT loop                     ; If not - repeat for
                               ;        next element
  ; on exit sum contained in r1
```

Subroutines

- A Branch and Link (BL) instruction is used to call a subroutine
- The return address is loaded into register R14, which acts as a link register
- When subroutines are nested, the contents of the link register must be saved on a stack by the subroutine.

  Register R13 is normally used as the pointer for this stack

- Return to the main code by MOV PC, R14

Example

Calling program

```
        LDR R1, N
        LDR R2, POINTER
        BL LISTADD
        STR R0, SUM
        .
        .
        .
```

Subroutine

```
LISTADD STMFD R13!, {R3, R14}     Save R3 and return address in R14 on stack,
                                  using R13 as the stack pointer
        MOV R0, #0
LOOP    LDR R3, [R2], #4
        ADD R0, R0, R3
        SUBS R1, R1, #1
        BGT LOOP
        LDMFD R13!, {R3, R15}     Restore R3 and load return address into PC (r15)
```

Example of finding minimum number subroutine

```
min routine:
  if( x<= y)  return x;
  else return y;
```

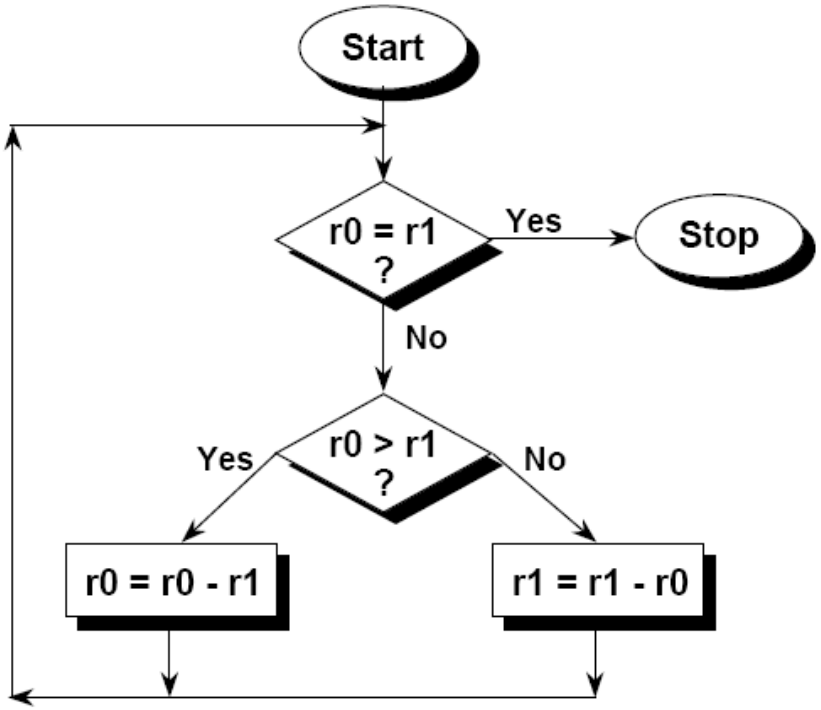
Solution

```
ldr  r0, x
ldr  r1, y
bal  min
str  r0, result

min:   cmp r0, r1
       movgt  r0, r1
       mov  PC, r14
```

Questions?

- What does this program do?
- Convert it into ARM assembly code?
- Convert it into ARM assembly using conditional execution?



Solution 1

```
gcd     cmp r0, r1        ;reached the end?
        beq stop
        blt less          ;if r0 > r1
        sub r0, r0, r1    ;subtract r1 from r0
        bal gcd
less    sub r1, r1, r0    ;subtract r0 from r1
        bal gcd
stop
```

Solution 2

```
gcd     cmp   r0, r1       ;if r0 > r1
        subgt r0, r0, r1   ;subtract r1 from r0
        sublt r1, r1, r0   ;else subtract r0 from r1
        bne   gcd          ;reached the end?
```

Byte-Sorting Program

```
for (j=n-1; j>0; j=j-1)
    {
    for (k=j-1; k>=0; k=k-1)
            {
            if (LIST[k]>LIST[j])
                    {
                    TEMP=LIST[k];
                    LIST[k]=LIST[j];
                    LIST[j]=TEMP;
                    }
            }
    }
```

Byte-Sorting Program

```
                ADR     R4,LIST     Load list pointer register R4
                LDR     R10,N       Initialize outer loop base
                ADD     R2,R4,R10   Register R2 to LIST+n
                ADD     R5,R4, #1   Load LIST+1 into R5
OUTER           LDRB    R0,[R2,# -1]!  Load LIST(j) into R0
                MOV     R3,R2       Initialize inner loop base
                                    register R3 to LIST+n-1
INNER           LDRB    R1,[R3, # -1]!  Load LIST(k) into R1
                CMP     R1,R0       Compare LIST(k) to LIST(j)
                                    If LIST(k)>LIST(j),
                STRGTB R1,[R2]      interchange LIST(k) and LIST(j)
                STRGTB R0,[R3]
                MOVGT   R0,R1       Move (new) LIST(j) into R0
                CMP     R3,R4       If k>0, repeat
                BNE     INNER       innerloop
                CMP     R2,R5       If j>1, repeat
                BNE     OUTER       outerloop
```

ARM instruction set

- Data Processing Instructions
- Load/Store Instructions
- Branch Instructions
- Control Instructions

Data Processing Instructions

- Move

Mnemonic	Definition	Op Mode bits [25:21]
MOV	*Move a 32-bit value into a register*	*1 1 0 1*
MVN	*Move the complement of the 32-bit value into a register*	*1 1 1 1*

- Arithmetic

Mnemonic	Definition	Op Mode bits [25:21]
ADD	*Add two 32-bit numbers*	*0 1 0 0*
ADC	*Add two 32-bit numbers with carry*	*0 1 0 1*
SUB	*Subtract two 32-bit numbers*	*0 0 1 0*
SBC	*Subtract two 32-bit numbers with carry*	*0 1 1 0*
RSB	*Reverse subtract two 32-bit numbers*	*0 0 1 1*
RSC	*Reverse subtract two 32-bit numbers with carry*	*0 1 1 1*

Data processing Instructions

- Logical operations

Mnemonic	Definition	Op Mode bits [25:21]
AND	*Bitwise AND of two 32-bit operands*	*0 0 0 0*
ORR	*Bitwise ORR of two 32-bit operands*	*1 1 0 0*
EOR	*Bitwise Exclusive OR of two 32-bit operands*	*0 0 0 1*
BIC	*Bitwise logical clear (AND NOT)*	*1 1 1 0*

- Compare operations

Mnemonic	Definition	Op Mode bits [25:21]
CMP	*Compare two 32-bit values*	*1 0 1 0*
CMN	*Compare negated*	*1 0 1 1*
TEQ	*Test two 32-bit numbers for equality*	*1 0 0 1*
TST	*Tests the bits of a 32-bit number (Logical AND)*	*1 0 0 0*

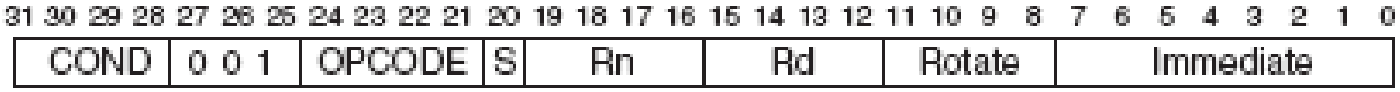
Data processing instructions

- Multiplications

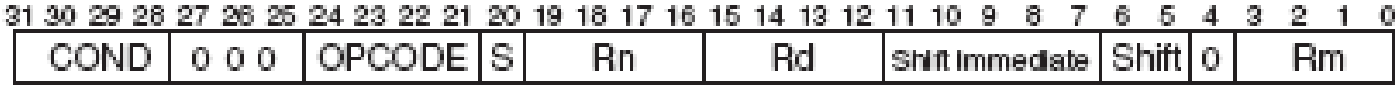
Mnemonic	Description	Syntax
MUL	*Multiply two 32-bit numbers, produce a 32-bit result: Rd = Rm * Rs*	`MUL{cond}{S} Rd, Rm, Rn`
MLA	*Multiply two 32-bit numbers, and add 3rd number for a 32-bit result: Rd = Rn + (Rm * Rs)*	`MLA{cond}{S} Rd, Rm, Rn, Rs`
UMULL	*Multiply two unsigned 32-bit numbers, produce an unsigned 64-bit resulted in two registers: [RdHi][RdLo] = Rm * Rs*	`UMULL{cond}{S} RdLo,RdHi,Rm,Rs`
UMLAL	*Multiply two unsigned 32-bit numbers and add an unsigned 64-bit number in two registers to produce an unsigned 64-bit resulted in two registers: [RdHi][RdLo] = [RdHi][RdLo] + Rm * Rs*	`UMLAL{cond}{S} RdLo,RdHi,Rm,Rs`
SMULL	*Multiply two signed 32-bit numbers, produce a signed 64-bit result in two registers*	`SMULL{cond}{S} RdLo,RdHi,Rm,Rs`
SMLAL	*Multiply two signed 32-bit numbers and add a signed 64-bit number in two registers to produce a signed 64-bit resulted in two registers: [RdHi][RdLo] = [RdHi][RdLo] + Rm * Rs*	`SMLAL{cond}{S} RdLo,RdHi,Rm,Rs`

Format of data processing instruction

Immediate operand



Immediate shift operand



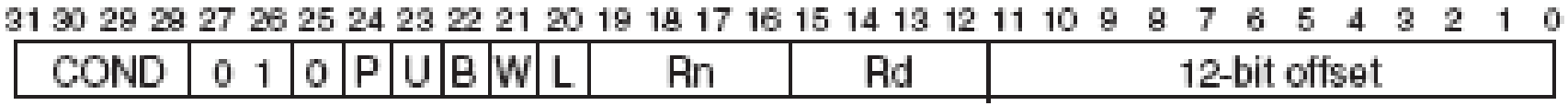
Register operand shift

Load/Store instructions

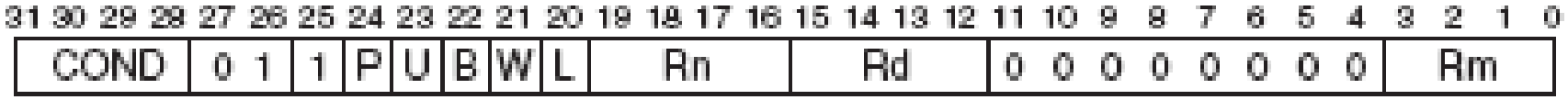
Mnemonic	Description	Syntax
LDR	*Load a register from a 32-bit memory word*	`LDR{cond} Rn,<address mode>`
LDRB	*Load a register from an 8-bit memory byte*	`LDRB{cond} Rn,<address mode>`
LDRH	*Load a register from an 16-bit memory half-word*	`LDRH{cond} Rn,<address mode>`
LDRSB	*Load a register from an 8-bit signed memory byte*	`LDRSB{cond} Rn,<address mode>`
LDRSH	*Load a register from a 16-bit signed memory half-word*	`LDRSH{cond} Rn,<address mode>`

Load/store instruction format

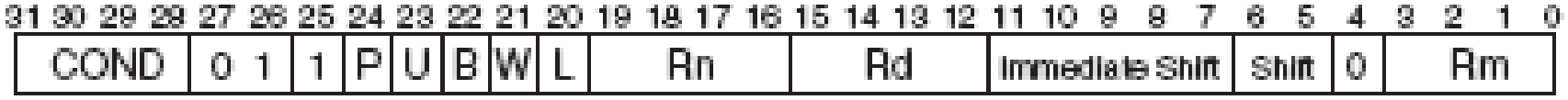
Immediate offset/Index



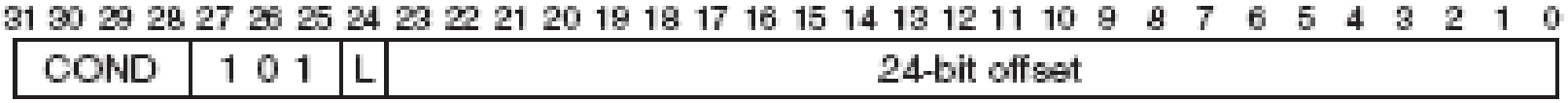
Register offset/Index



Scaled register offset/Index

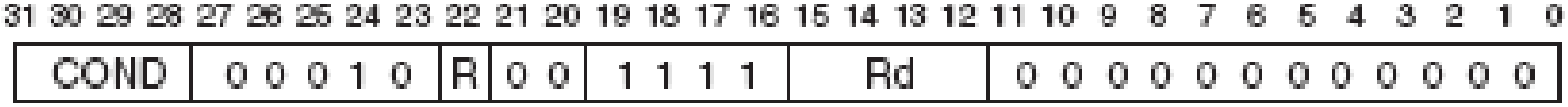


Branch instruction format



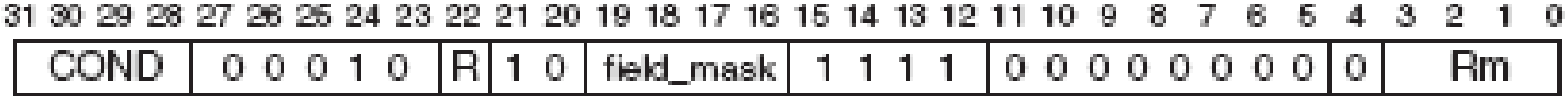
Control instruction format

MSR

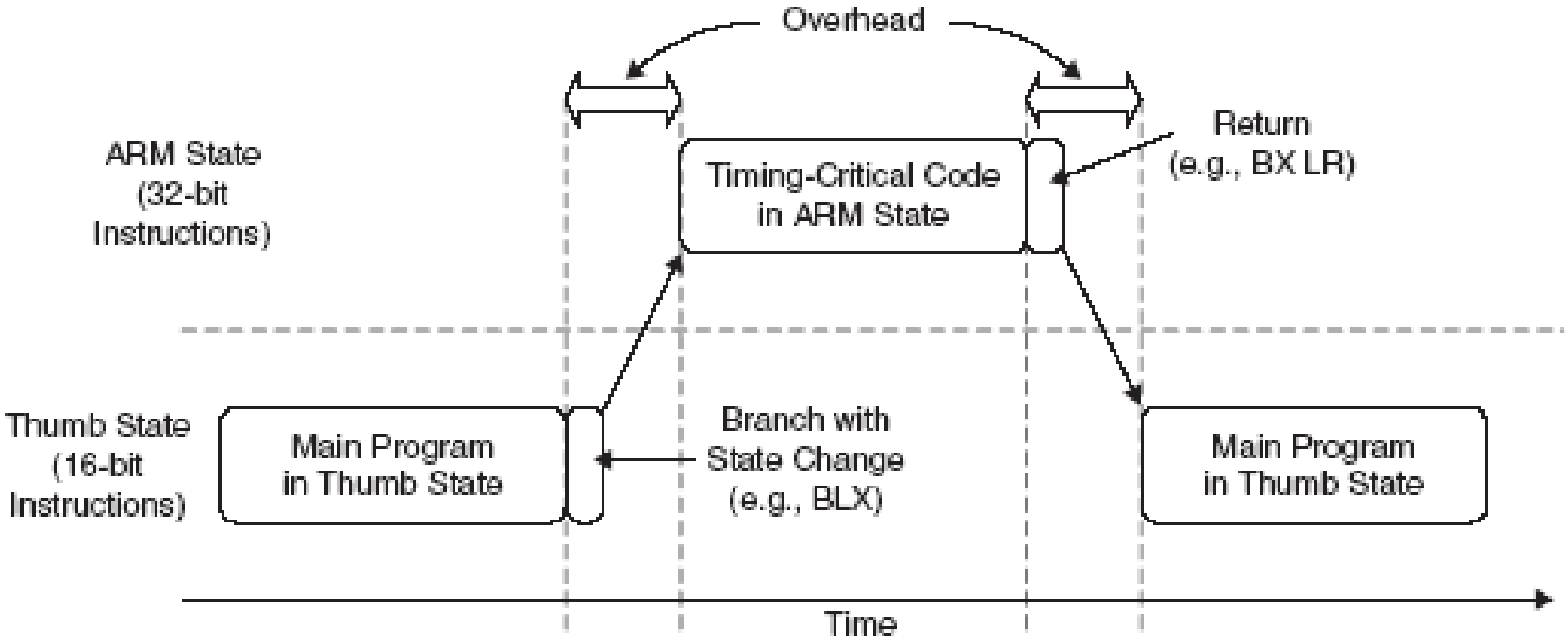


MSR Immediate Form

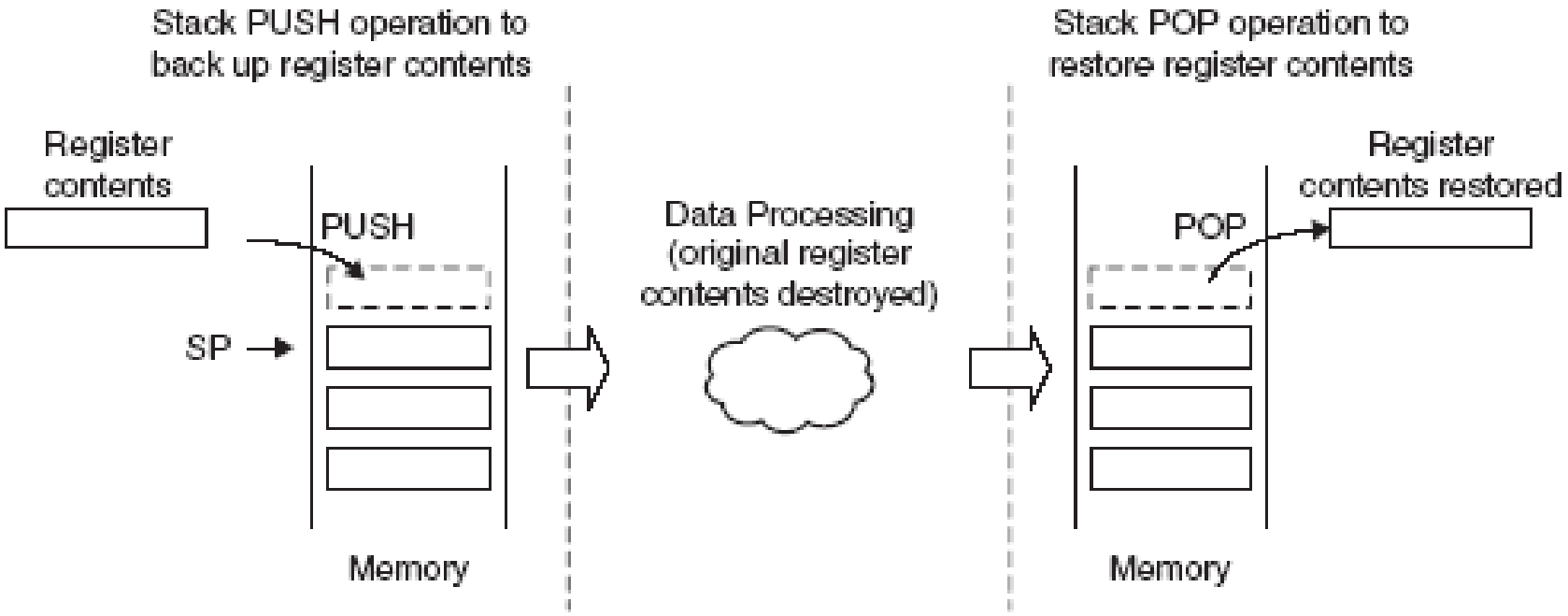
MSR Register Form



Switching between ARM code and thumb mode



Basic concept of Stack



Push-Pop instruction

```
PUSH        {R0}   ; R13=R13-4, then Memory[R13] = R0
POP         {R0}   ; R0 = Memory[R13], then R13 = R13+4
```

Sub-routine call

```
subroutine_1
        PUSH            {R0-R7, R12, R14}   ; Save registers
        …                                   ; Do your processing
        POP             {R0-R7, R12, R14}   ; Restore registers
        BX              R14                 ; Return to calling function
```

For BX instruction, If bit 0 of r14 is 1, then execute Thumb instructions. Otherwise, execute ARM instructions.

Using Link Register

```
main    ; Main program
        …
        BL   function1  ; Call function1 using Branch with Link
                        ; instruction.
                        ; PC = function1 and
                        ; LR = the next instruction in main
        …
function1
        …               ; Program code for function 1
        BX   LR         ; Return
```

Stack operation

Main Program

```
    ...
; R0 = X, R1 = Y, R2 = Z
BL    function1
```

Subroutine

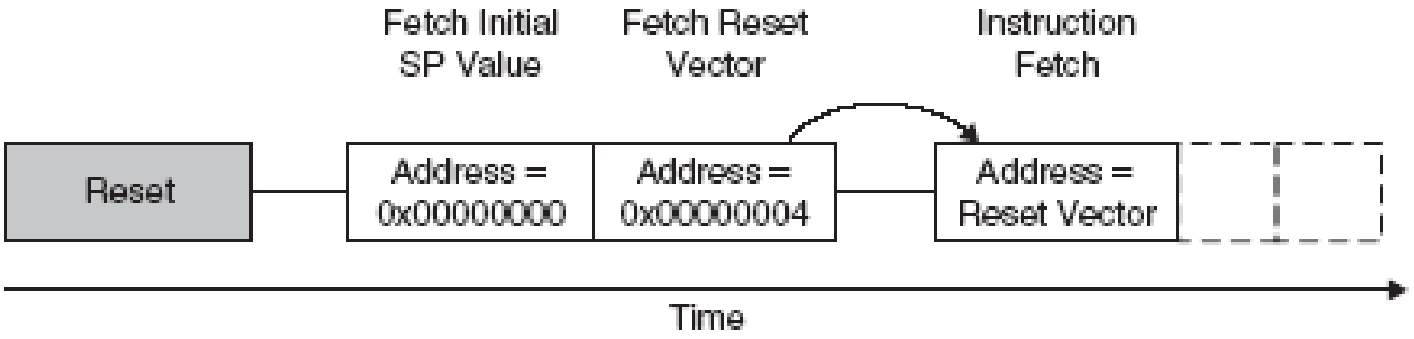
```
function1
    PUSH    {R0}  ; store R0 to stack & adjust SP
    PUSH    {R1}  ; store R1 to stack & adjust SP
    PUSH    {R2}  ; store R2 to stack & adjust SP
    ...           ; Executing task (R0, R1 and R2
                  ; could be changed)
    POP     {R2}  ; restore R2 and SP re-adjusted
    POP     {R1}  ; restore R1 and SP re-adjusted
    POP     {R0}  ; restore R0 and SP re-adjusted
    BX      LR    ; Return
```

```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

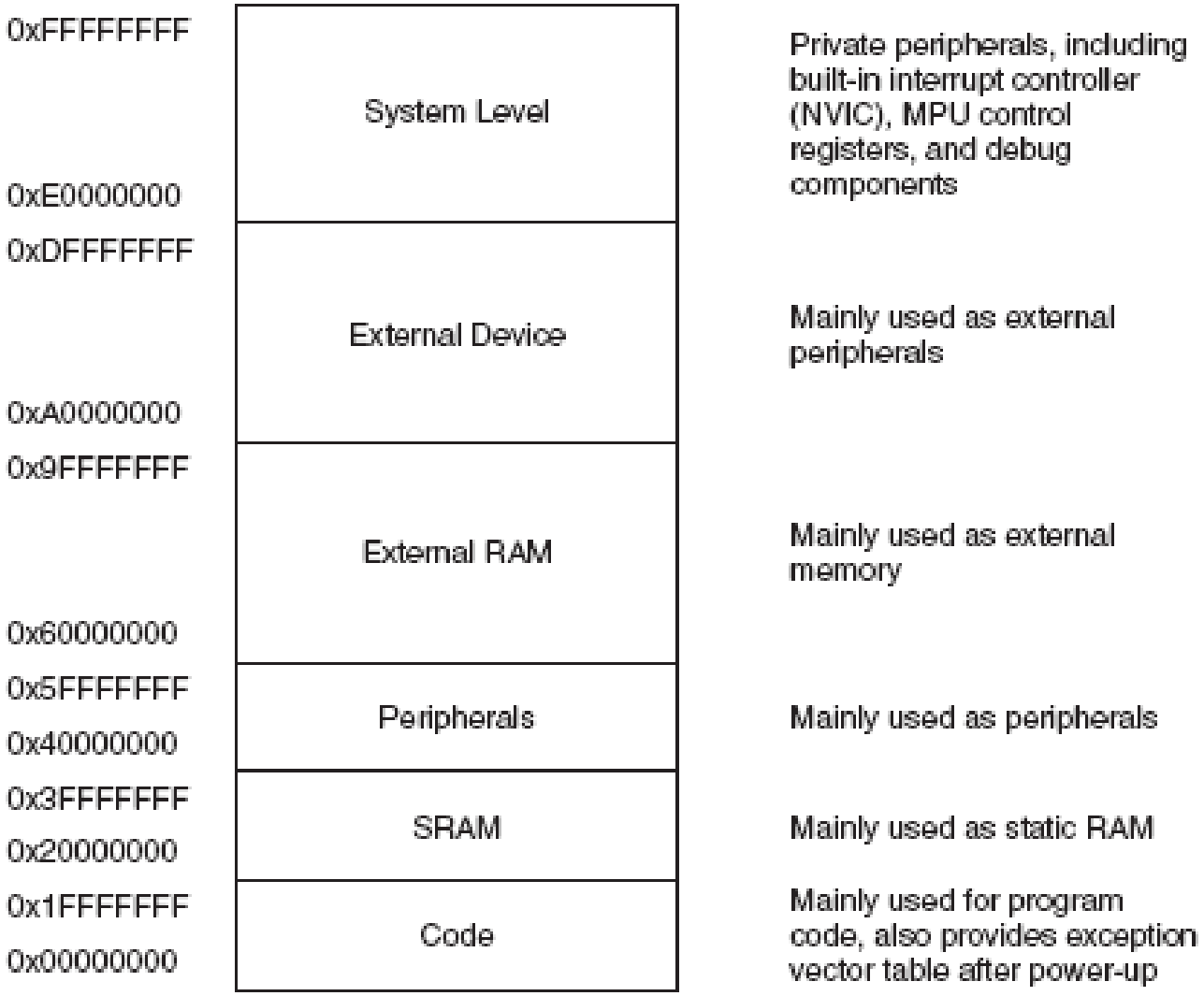
ARM System vectors

Exception Vector	Address
Reset	*0x00000000*
Undefined Instructions	*0x00000004*
Software Interrupt	*0x00000008*
Prefetch Abort	*0x0000000C*
Data Abort	*0x00000010*
Reserved	*0x00000014*
Interrupt Request	*0x00000018*
Fast Interrupt Request	*0x0000001C*

Reset sequence



Memory map


0xFFFFFFFF
System Level
0xE0000000
0xDFFFFFFF
External Device
0xA0000000
0x9FFFFFFF
External RAM
0x60000000
0x5FFFFFFF
Peripherals
0x40000000
0x3FFFFFFF
SRAM
0x20000000
0x1FFFFFFF
Code
0x00000000
Private peripherals, including built-in interrupt controller (NVIC), MPU control registers, and debug components
Mainly used as external peripherals
Mainly used as external memory
Mainly used as peripherals
Mainly used as static RAM
Mainly used for program code, also provides exception vector table after power-up