ARM

The ARM Architecture

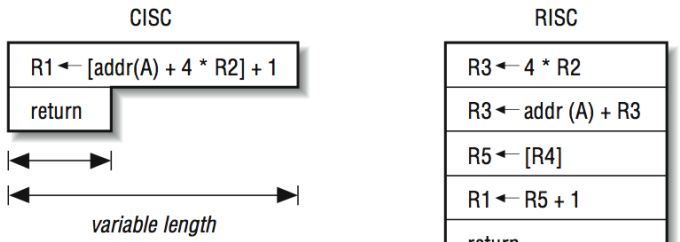
ARM Ltd

- **Founded in November 1990**
  - Spun out of Acorn Computers
- **Designs the ARM range of RISC processor cores**
- **Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers.**
  - ARM does not fabricate silicon itself
- **Also develop technologies to assist with the design-in of the ARM architecture**
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc

ARM

CISC vs RISC

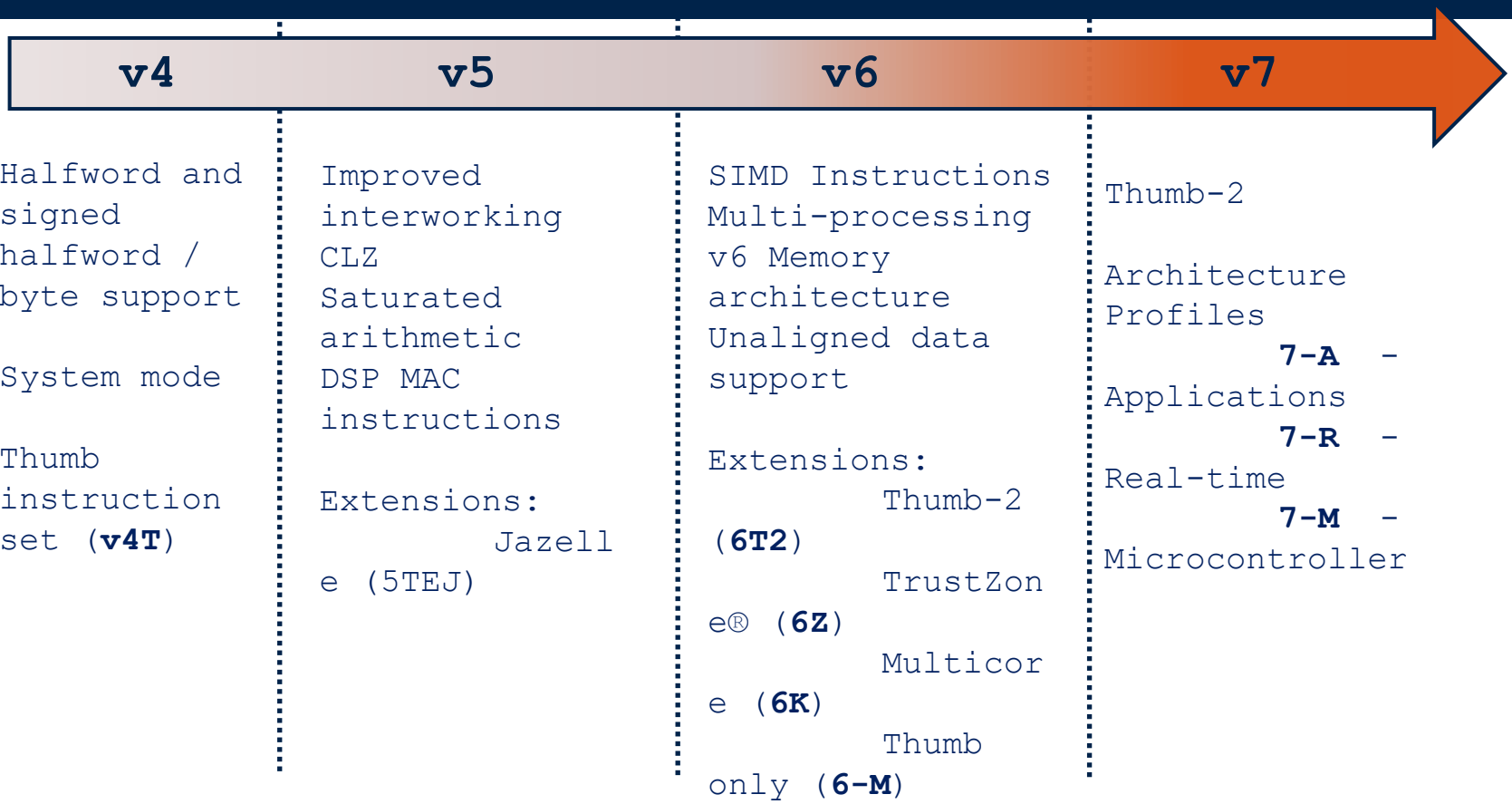
CISC	RISC
complex instructions	simple instructions
main focus is Hardware	main focus is Software
complexity lies in processor	complexity lies in compiler
Multiple clock cycle	single clock cycle
Transistors are used to store complex instructions	Transistors are used for storing memory
CISC has100-300 minimum.	uses Few instructions(30-40)
8 to 10 addressing modes.	few addressing modes.
Variable size/length instructions	fixed size/length instructions



ARM Connected Community – 900+

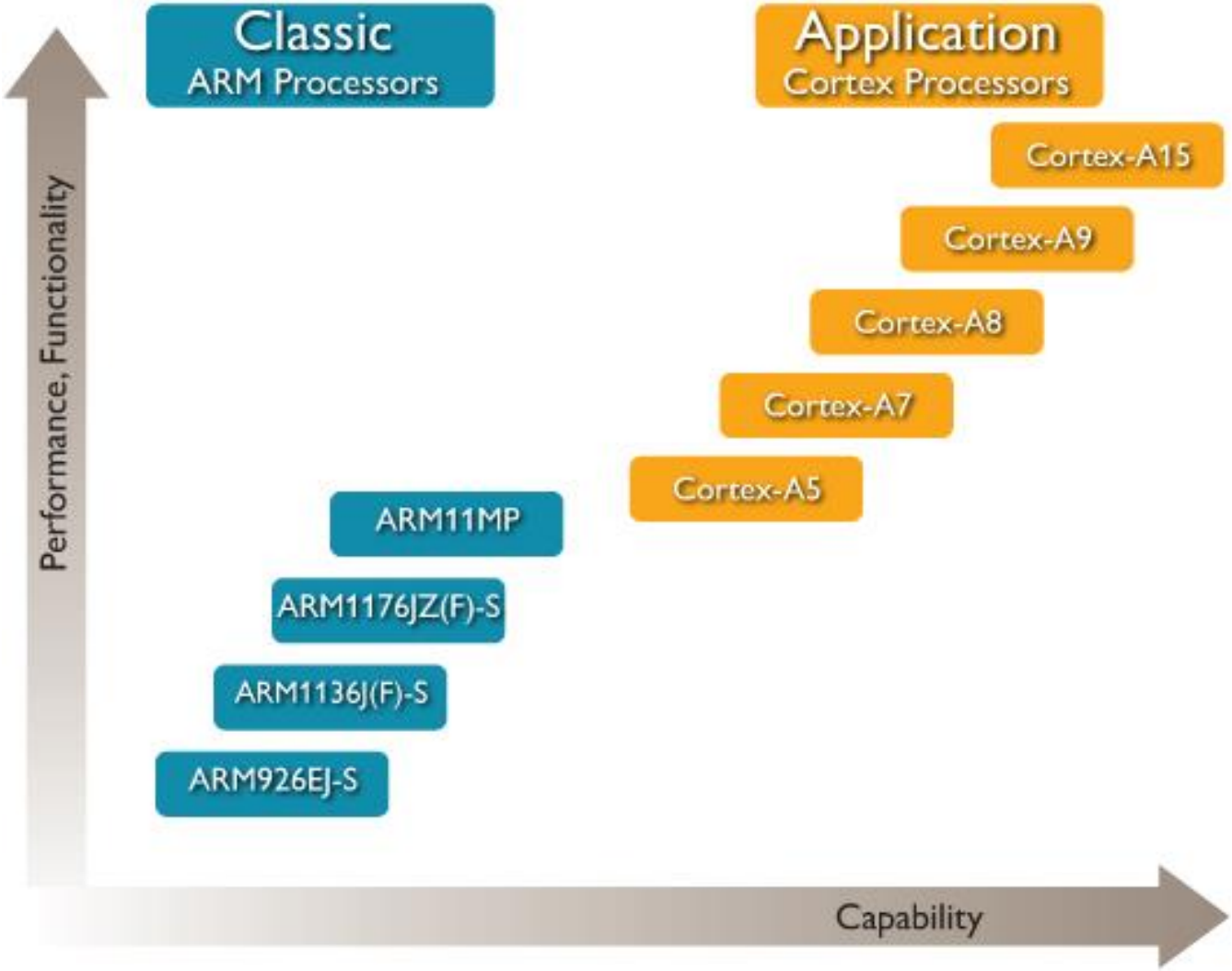
Connect, Collaborate, Create – accelerating innovation

ARM Development of the ARM Architecture



- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

Application Processors



Architecture ARMv7 profiles

- **Application profile (ARMv7-A)**
  - Memory management support (MMU)
  - Highest performance at low power
    - Influenced by multi-tasking OS system requirements
  - TrustZone and Jazelle-RCT for a safe, extensible system
  - e.g. Cortex-A5, Cortex-A9

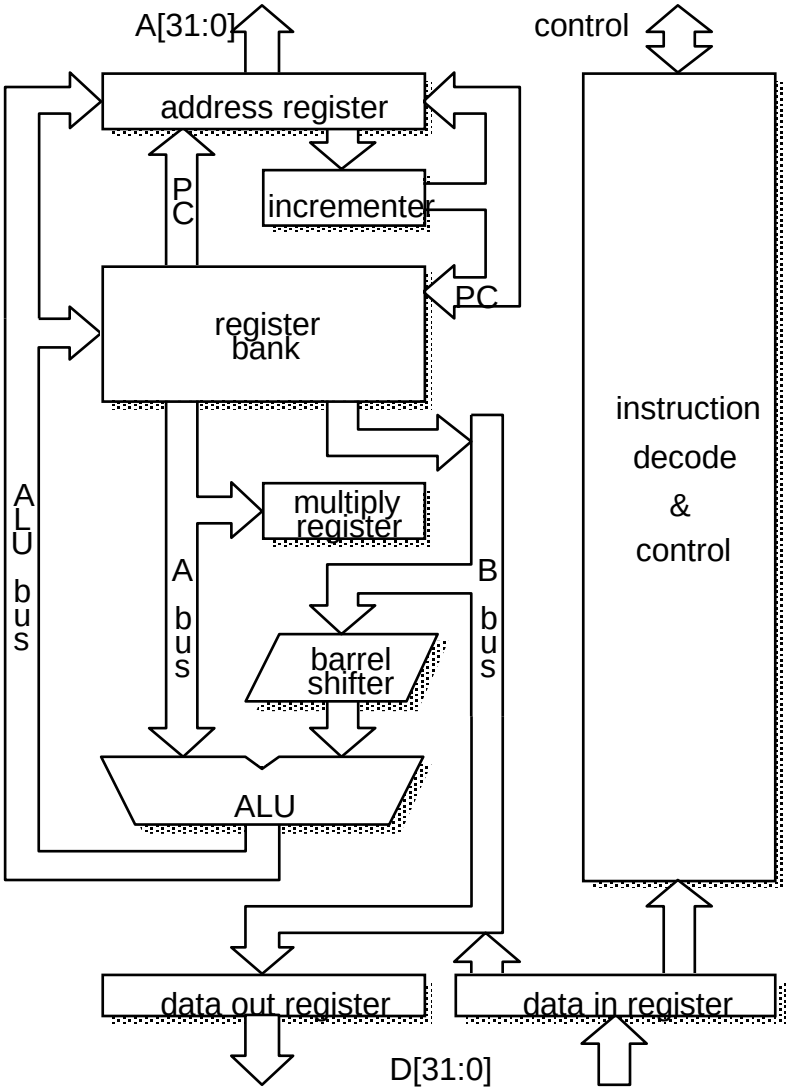
- **Real-time profile (ARMv7-R)**
  - Protected memory (MPU)
  - Low latency and predictability ‘real-time’ needs
  - Evolutionary path for traditional embedded business
  - e.g. Cortex-R4

- **Microcontroller profile (ARMv7-M, ARMv7E-M, ARMv6-M)**
  - Lowest gate count entry point
  - Deterministic and predictable behavior a key priority
  - Deeply embedded use
  - e.g. Cortex-M3

ARM Powered Products

ARM organization

- **Register file –**
  - 2 read ports, 1 write port +
    1 read, 1 write port reserved for r15 (pc)
- **Barrel shifter – shift or rotate one operand for any number of bits**
- **ALU – performs the arithmetic and logic functions required**
- **Memory address register + incrementer**
- **Memory data registers**
- **Instruction decoder and associated control logic**



Data Sizes and Instruction Sets

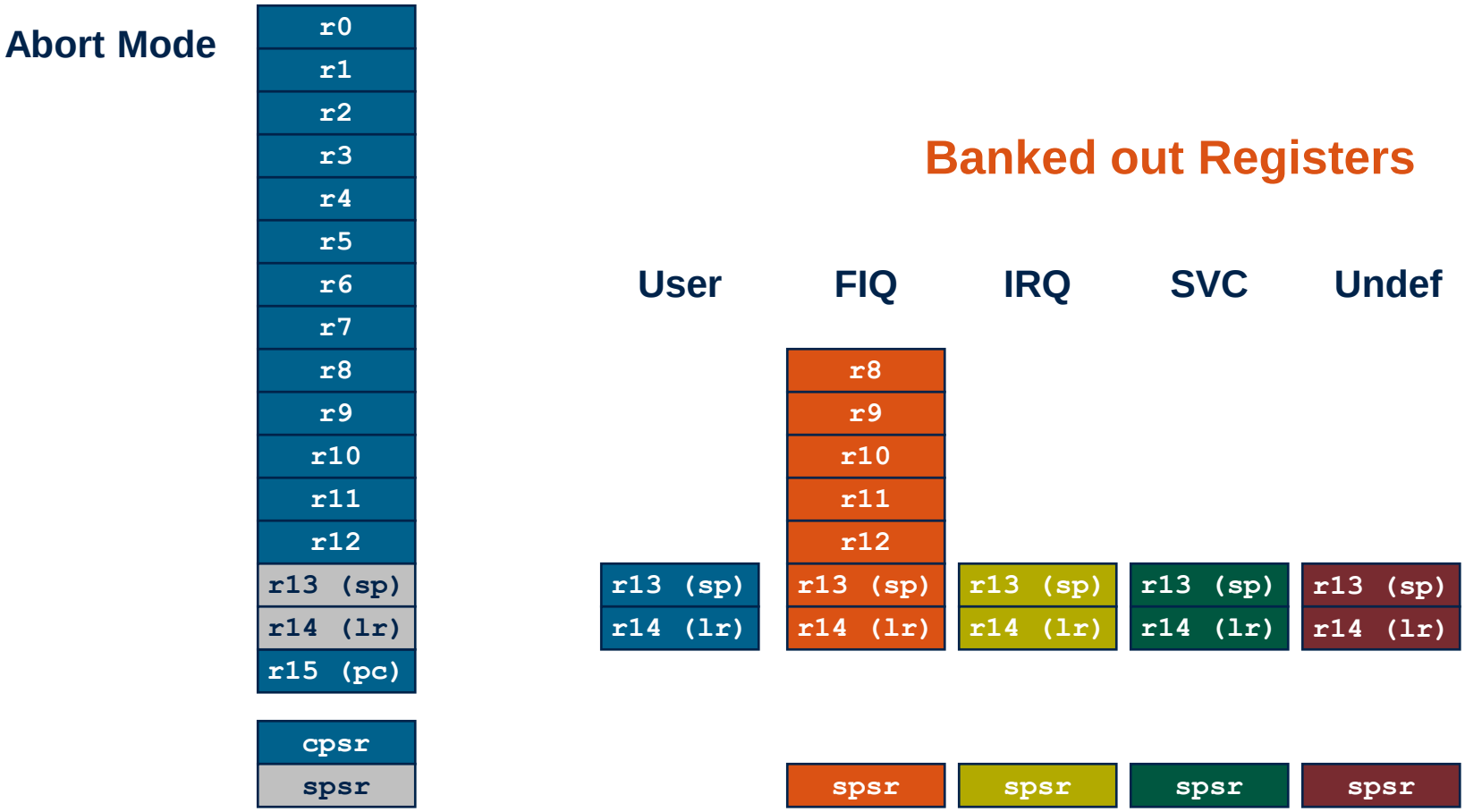
- **The ARM is a 32-bit architecture.**
- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
- **Most ARM's implement two instruction sets**
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
- **Jazelle cores can also execute Java bytecode**

Processor Modes

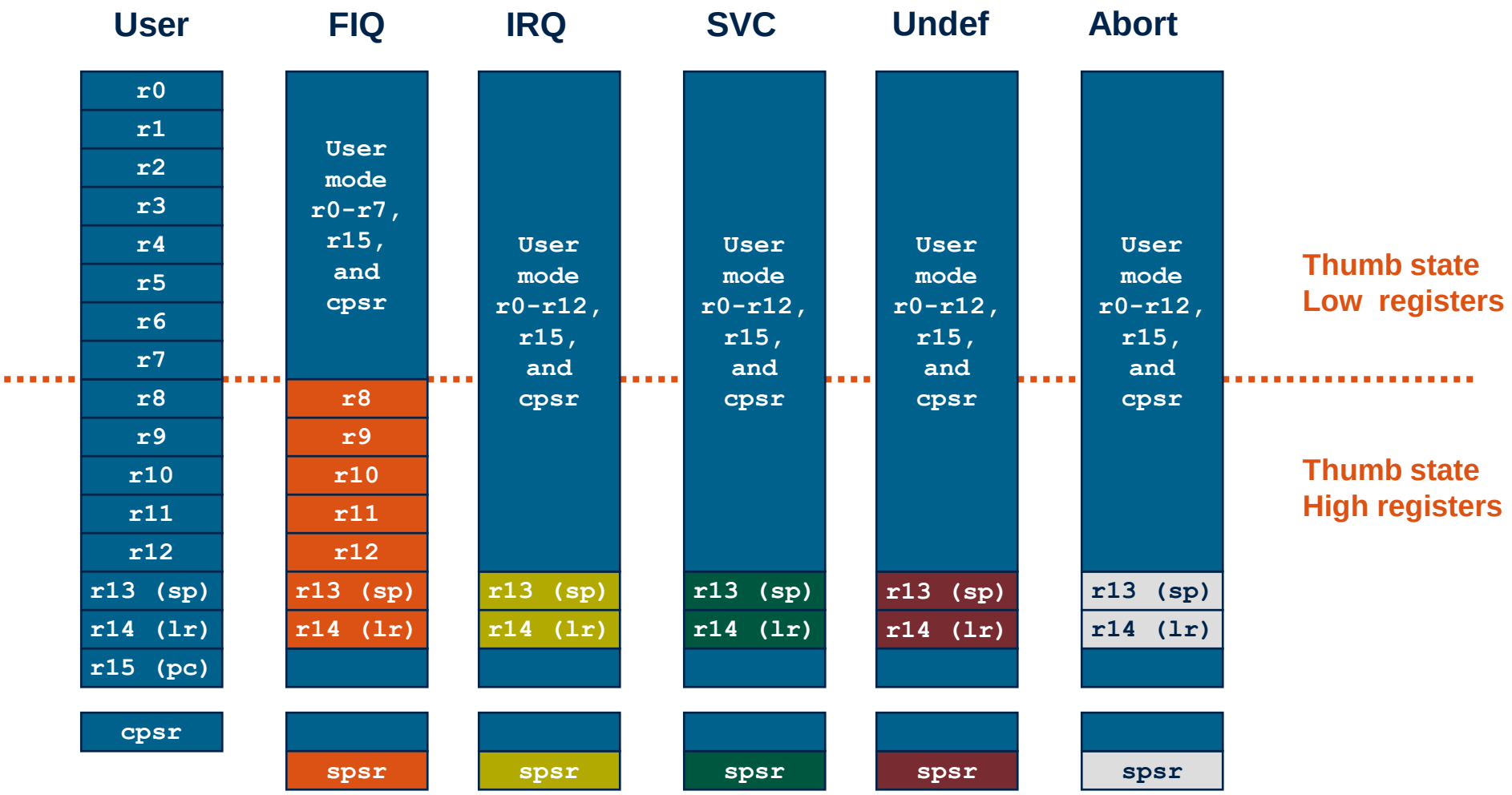
- **The ARM has seven basic operating modes:**
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

The ARM Register Set

Current Visible Registers



Register Organization Summary



Note: System mode uses the User mode register set

Banked registers

- Duplicates of some of the registers in the range R8 through R14 are provided for each of the processor modes other than the User and System modes
- Banked registers make context switches between the modes more efficient by avoiding register save/restore operations on such switches

The Registers

- **ARM has 37 registers all of which are 32-bits long.**
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers

- **The current processor mode governs which of several banks is accessible.**

Each mode can access

- a particular set of r0-r12 registers
- a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
- the program counter, r15 (pc)
- the current program status register, cpsr

Privileged modes (except System) can also access

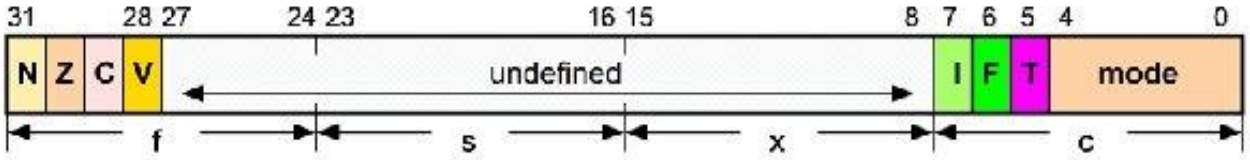
- a particular spsr (saved program status register)

Registers across CPU modes

usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Program Status Registers

Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- **Mode bits**

10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

Never change T directly (use BX instead)

Changing T in CPSR will lead to unexpected behavior due to pipelining

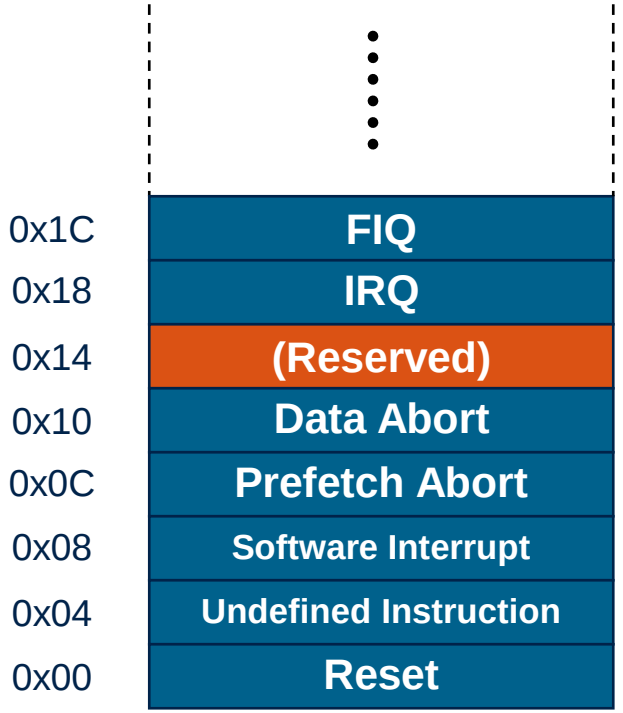
Tip: Don't change undefined bits.

This allows for code compatibility with newer ARM processors

Exception Handling

- **When an exception occurs, the ARM:**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- **To return, exception handler needs to:**
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

  This can only be done in ARM state.



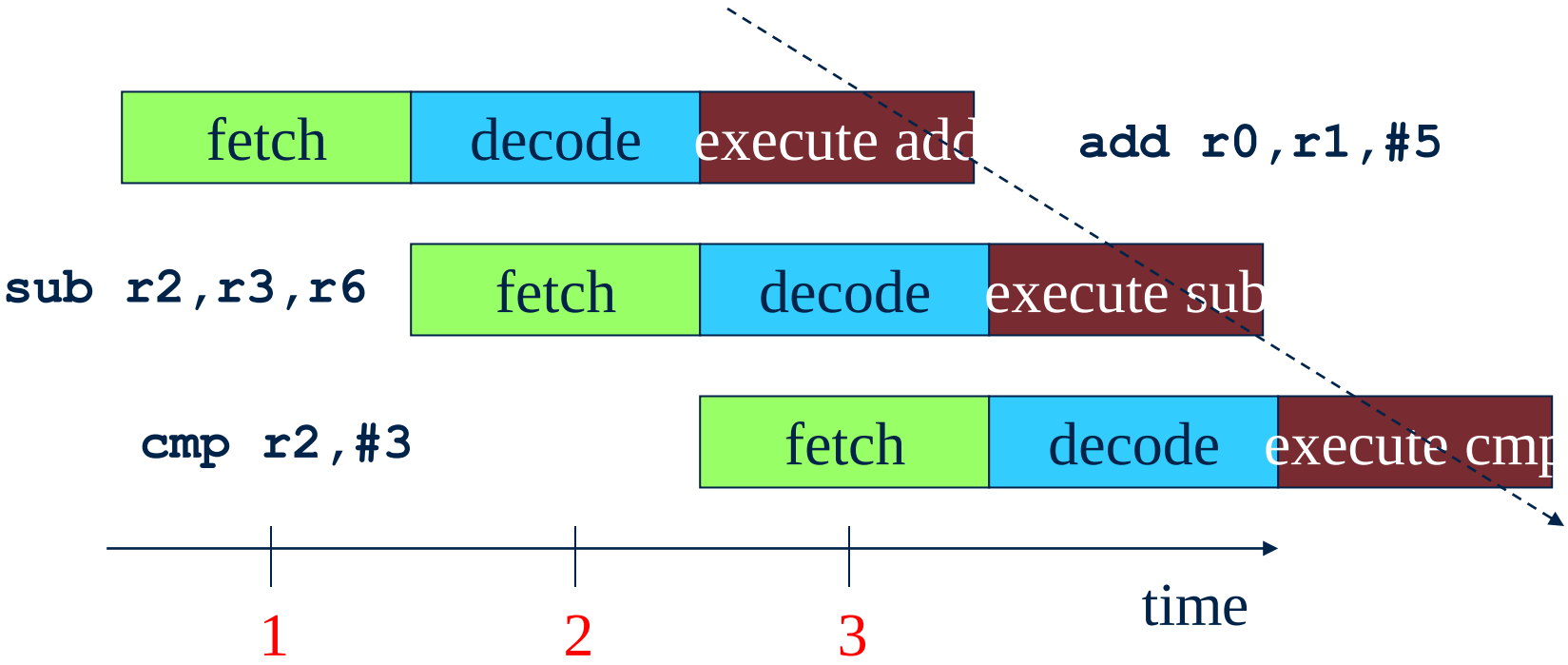
Vector Table

Vector table can be at **`0xFFFF0000`** on ARM720T and on ARM9/10 family devices

Three-stage pipeline

- **Fetch**
  - the instruction is fetched from memory and placed in the instruction pipeline
- **Decode**
  - the instruction is decoded and the datapath control signals prepared for the next cycle; in this stage the instruction owns the decode logic but not the datapath
- **Execute**
  - the instruction owns the datapath; the register bank is read, an operand shifted, the ALU register generated and written back into a destination register

ARM single-cycle instruction pipeline



Features of ARM instruction set

- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple registers at once
- Possible to combine shift and ALU operations in a single instruction

ARM

3-address instructions

Syntax of Instructions: 1 2, 3, 4 where:

1) instruction by name

2) operand getting result (“destination”)

3) 1st operand for operation (“source1”)

4) 2nd operand for operation (“source2”)

Syntax is rigid (for the most part):

1 operator, 3 operands

- How do the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
ADD r0, r1, r2 ; a = b + c
ADD r0, r0, r3 ; a = a + d
SUB r0, r0, r4 ; a = a - e
```

- Notice: A single line of C may break up into several lines of ARM.

Instruction set

- Data processing
- Data movement (memory access)
- Flow control

Data processing Instructions

- **Consist of :**
  - Arithmetic: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Logical: `AND` `ORR` `EOR` `BIC`
  - Comparisons: `CMP` `CMN` `TST` `TEQ`
  - Data movement: `MOV` `MVN`

- **These instructions only work on registers, NOT memory.**

- **Syntax:**

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- **Second operand is sent to the ALU via barrel shifter.**

Almost all ARM instructions have a condition field which allows it to be executed conditionally.

```
MOV<cc><S>  Rd, <operands>

MOVCS R0, R1 @ if carry is set
             @ then R0:=R1

MOVS  R0, #0 @ R0:=0
             @ Z=1, N=0
             @ C, V unaffected
```

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*v*erflow *S*et	VC	O*v*erflow *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	*Pl*us (Positive)	MI	*Mi*nus (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

Instructions

MVN R*d*, R*m* or #value

loads the bit-complement of [R*m*] or value into R*d*

Instructions

Implementing Shift and Rotate instructions:

e.g., MOV R*i*, R*j*, LSL #4

See the remaining slides for explanations

The Barrel Shifter

LSL : Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right

Division by a power of 2

ROR: Rotate Right

Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended

Single bit rotate with wrap around
from CF to MSB

LSL : Logical Left Shift


```
MOV   R0, R2, LSL #2 @ R0:=R2<<2
                     @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
       R2=0x00000030
```

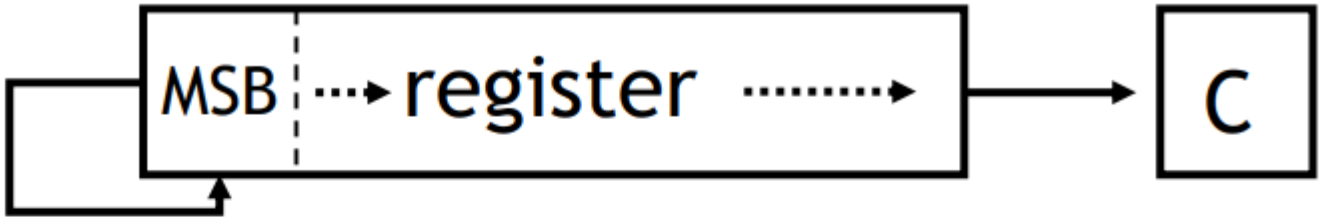
0000....1100 0000

LSR : Logical Shift Right


```
MOV   R0, R2, LSR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
       R2=0x00000030
```

0000 ... 0000 1100

ASR: Arithmetic Right Shift




```
MOV   R0, R2, ASR #2 @ R0:=R2>>2
                     @ R2 unchanged

Example: 1010 0...0 0011 0000

Before R2=0xA0000030

After  R0=0xE800000C
       R2=0xA0000030
```

ROR: Rotate Right


```
MOV   R0, R2, ROR #2 @ R0:=R2 rotate
                     @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
       R2=0x00000031
```

RRX: Rotate Right Extended

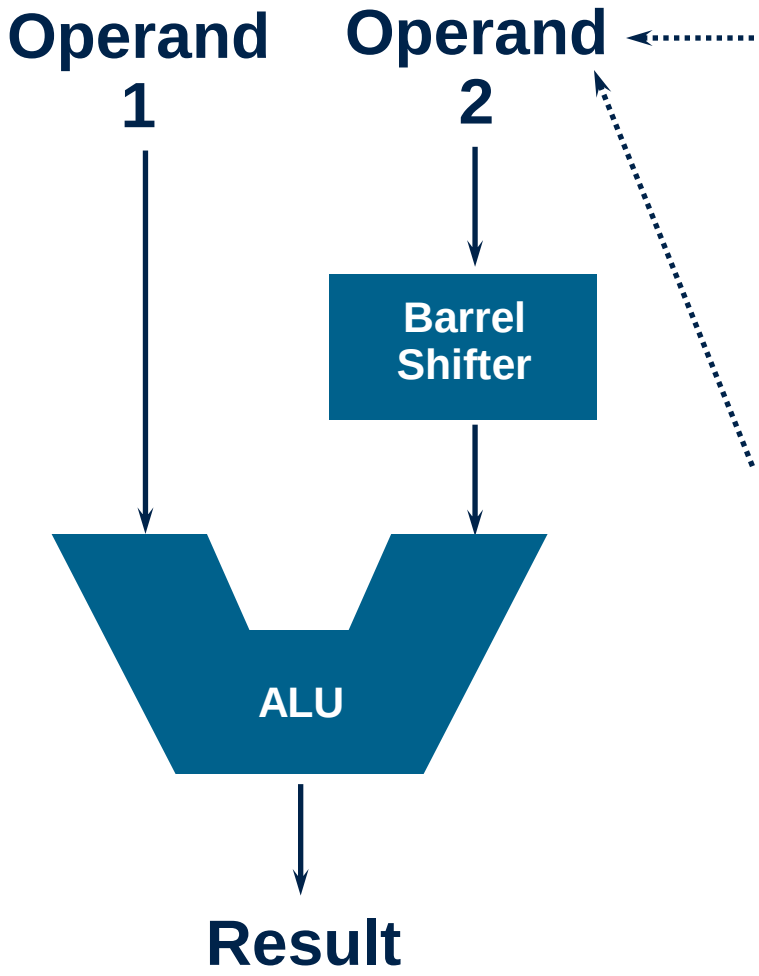

```
MOV   R0, R2, RRX      @ R0:=R2 rotate
                       @ R2 unchanged

Example: 0...0 0011 0001      SINGLE BIT ROTATE

Before R2=0x00000031, C=1

After  R0=0x80000018, C=1
       R2=0x00000031
```

Using the Barrel Shifter: The Second Operand



Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

Add/Subtract instructions

1. ADD r1, r2, r3; r1=r2+r3
2. ADC r1, r2, r3; r1=r2+r3+ C(arry Flag)
3. SUB r1, r2,r3; r1=r2-r3
4. SUBC r1, r2, r3; r1=r2-r3 +C -1
5. RSB r1, r2, r3; r1= r3-r2;
6. RSC r1, r2, r3; r1=r3-r2 +C -1

Add/Subtract instructions

Assembly language format is

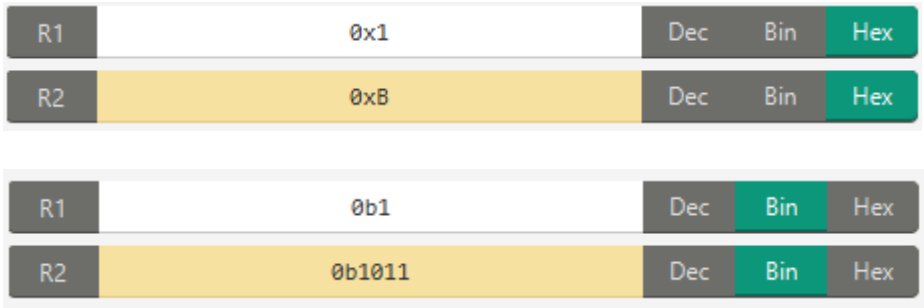
OP Rd, Rn, Rm or #offset

```
mov     r1, #1
add     r2, r1, #10
```

ADD R0, R2, R4

performs

R0 ← [R2] + [R4]



SUB R0, R3, #17

performs

R0 ← [R3] – 17

(immediates are unsigned values in the range 0 to 255)

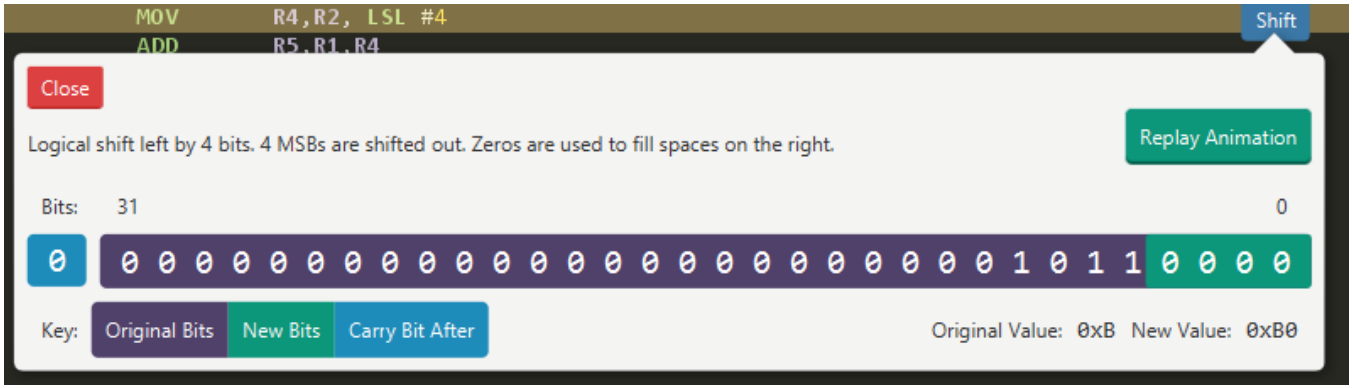
Add/Subtract instructions

The second source operand can be shifted or rotated before being used

```
ADD    R0, R1, R5, LSL #4
```

```
Reset to continue editing code
1        mov     r1, #1
2        add     r2, r1, #10
3        add     r3,r1,r2
4        mov     r4, r2, lsl #4
5        add     r5, r1, r4
6
7
```

R0	0x0	Dec	Bin	Hex
R1	0x1	Dec	Bin	Hex
R2	0b1011	Dec	Bin	Hex
R3	0xC	Dec	Bin	Hex
R4	0xB0	Dec	Bin	Hex
R5	0xB1	Dec	Bin	Hex

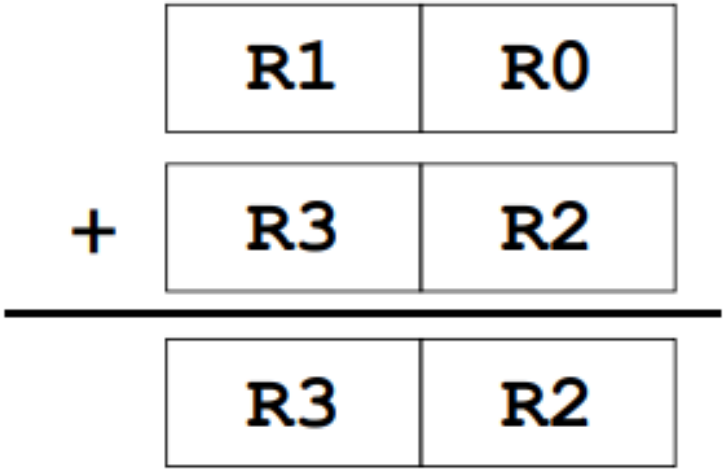


Add/Subtract instructions

Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

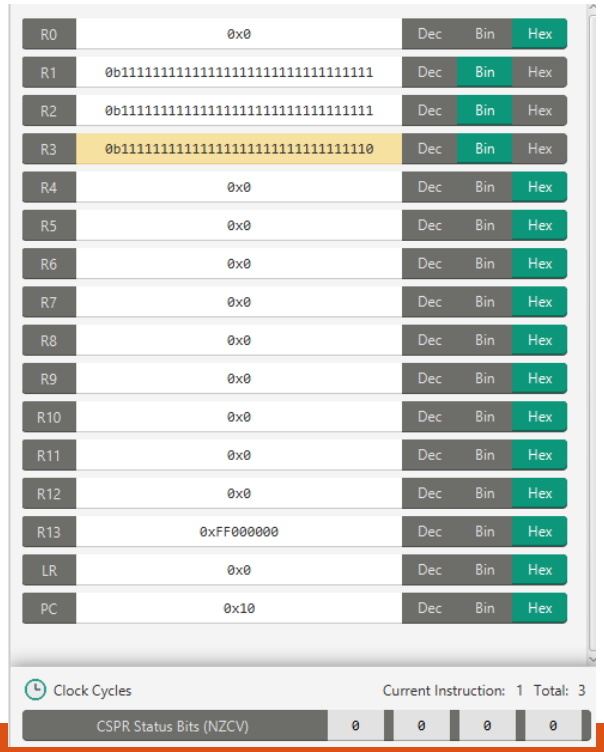
```
ADDS   R2, R2, R0
ADC    R3, R3, R1
```



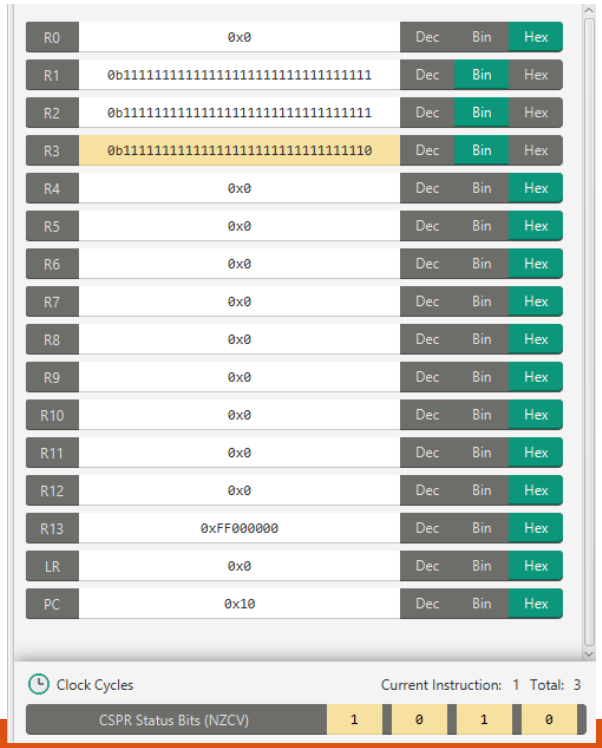
Add/Subtract instructions

- **Adding 64-bit operands**

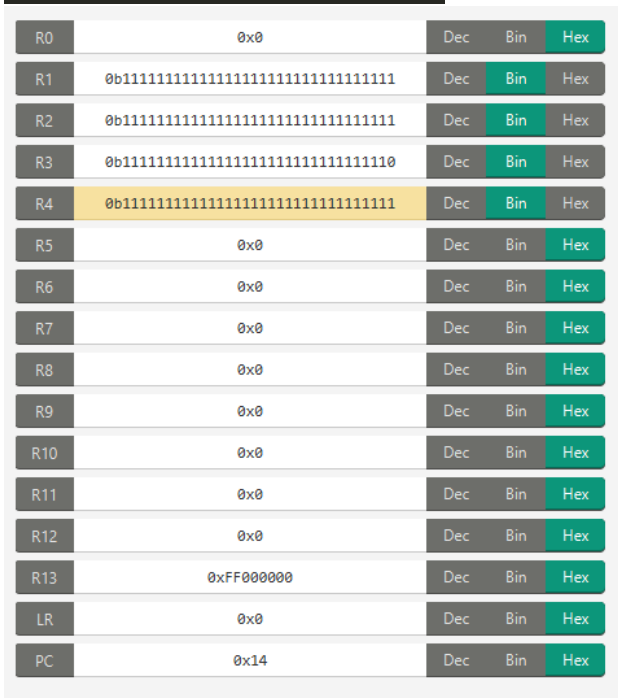
```
MOV   R1,#0XFFFFFFFF
MOV   R2,#0XFFFFFFFF
ADD   R3,R1,R2
```




```
MOV   R1,#0XFFFFFFFF
MOV   R2,#0XFFFFFFFF
ADDS  R3,R1,R2
```




```
MOV   R1,#0XFFFFFFFF
MOV   R2,#0XFFFFFFFF
ADDS  R3,R1,R2
ADC   R4,R1,R2
```



Multiply

- **Syntax:**
  - MUL{<cond>}{S} Rd, Rm, Rs — Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn — Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs — RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs — RdHi,RdLo := (Rm*Rs)+RdHi,RdLo

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2    ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3    ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Instructions

MUL R0, R1, R2 performs
R0←[R1] × [R2]

The low-order 32 bits of the 64-bit product are written into R0

For 2's-complement numbers, the value in R0 is correct if the product fits into 32 bits

Instructions

MLA R0, R4, R5, R6 performs
R0 ← ([R4] × [R5]) + [R6]

This Multiply-Accumulate instruction is useful in signal-processing applications

Other versions of MUL and MLA generate 64-bit products

Logical

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	$Rd = Rn \,\hat{}\, N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \,\&\, \sim N$

Logical

AND R*d*, R*n*, R*m*

performs the bit-wise logical AND of the operands in registers R*n* and R*m* and writes the result into register R*d*

ORR (bit-wise logical OR)
EOR (bit-wise logical XOR) are also provided

Logical

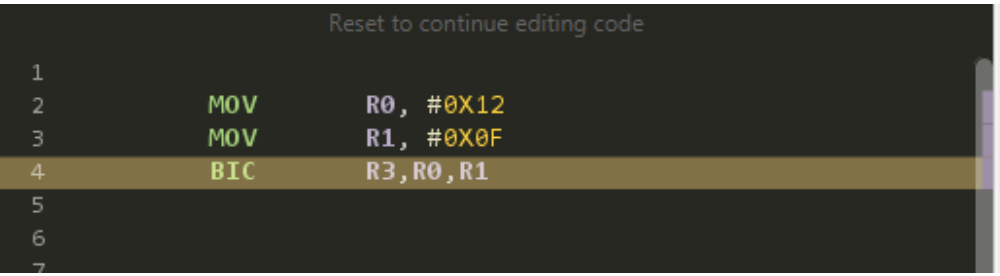
The Bit Clear instruction, BIC, is closely related to the AND instruction

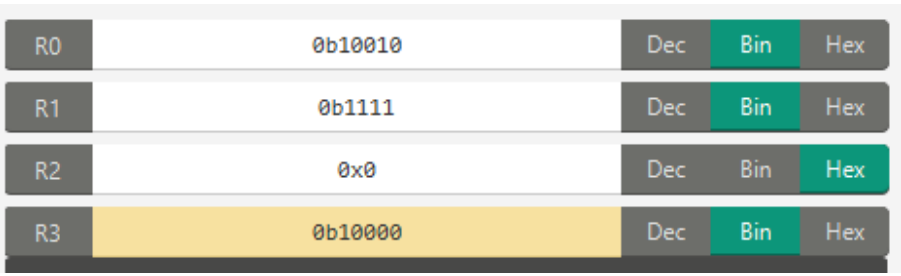
The bits of R*m* are complemented before they are ANDed with the bits of R*n*

If R0 contains the hexadecimal pattern 02FA62CA, and R1 contains 0000FFFF,

BIC R0, R0, R1, R1 = FFFF0000

- **results in 02FA0000 being written into R0**

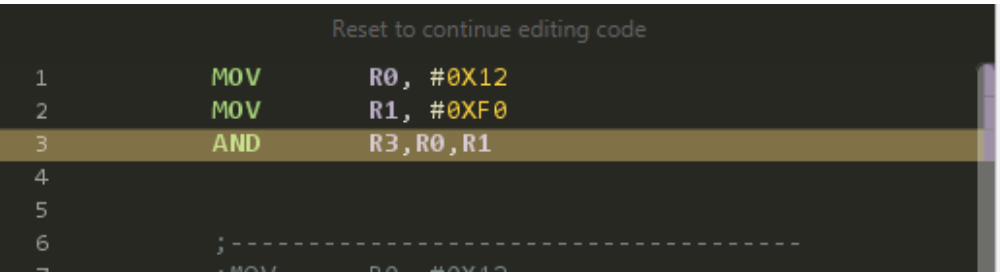


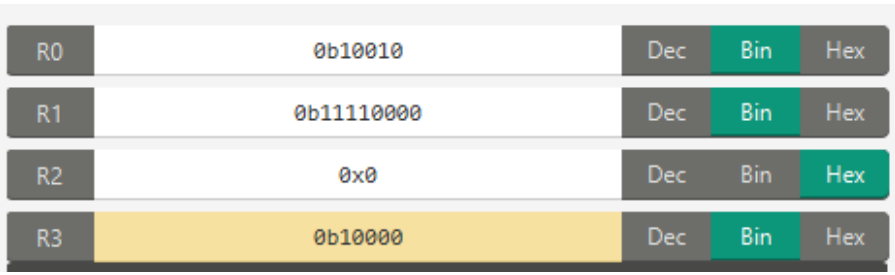


Logical: MASKING

If R0 contains the hexadecimal pattern 02FA62CA, and R1 contains 0000FFFF,

- **AND R0, R0, R1**
- **02FA62CA AND 0000FFFF, RESULT = 000062CA**
- **[R0] = 000062CA**
- **MASKING**





- **Test:**

TST R*n*, R*m* or #value
performs bit-wise logical AND of the two operands, then sets condition code flags

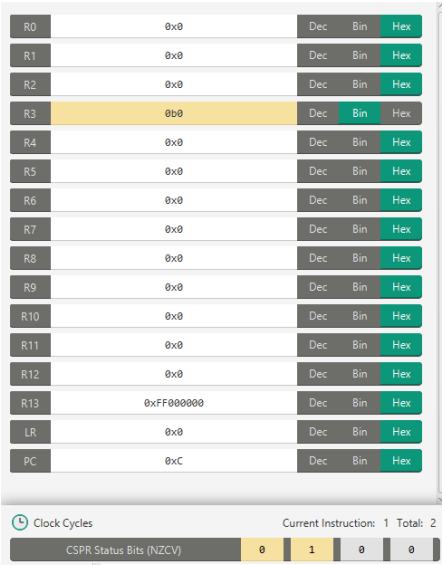
TST R3, #1
sets Z ← 1 if low-order bit of R3 is 0
sets Z ← 0 if low-order bit of R3 is 1

(useful for checking status bits in I/O devices)

TST R*n*, R*m* or #value

- IF R3 = 0000H
- AND WITH #1, MEANS AND WITH 0001H
- 0000H AND 0001H = 0000H, ZERO SET

```
MOV     R3, #0X0
TST     R3,#1
```



- IF R3= 0001H
- AND WITH 0001H
- 0001 AND 0001H = 0001H, ZERO IS RESET

```
MOV     R3, #0X1
TST     R3,#1
```

TEQ

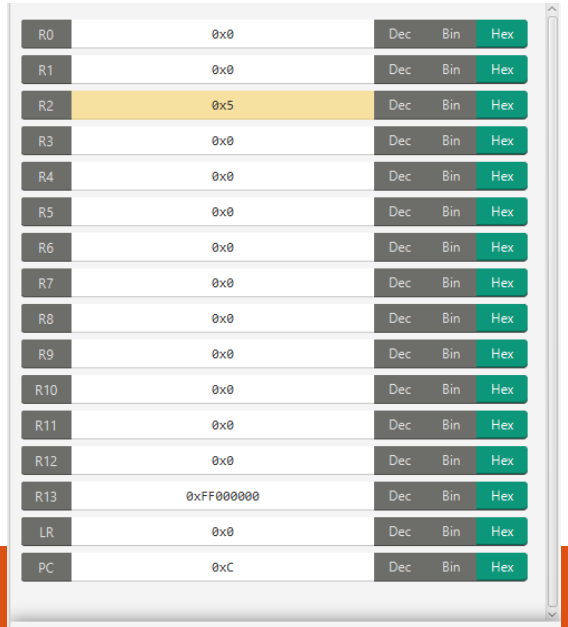
- **Test:**

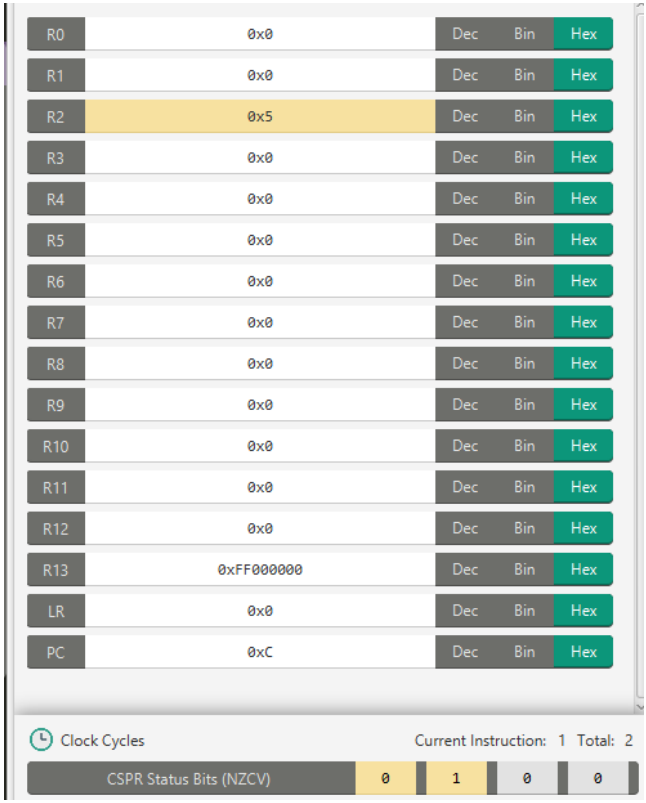
TEQ Rn, Rm or #value
performs bit-wise logical XOR of the two operands, then sets condition code flags

```
MOV     R2,#5
TEQ     R2,#5
```

sets Z ← 1 if R2 contains 5
sets Z ← 0 otherwise

```
MOV     R2,#5
TEQ     R2,#6
```





COMPARE

- **Compare:**

  CMP R*n*, R*m*

  performs

  [R*n*] - [R*m*]

  and updates condition code flags based on the result

- ***IF [RN] > [RM], ZERO= 0, N =0***
- ***IF [RN] = [RM], ZERO= 1, N =0***
- ***IF[RN] < [RM], ZERO = 0, N =1***

CMP Rn, Rm performs [Rn] - [Rm]

```
MOV   R1,#5
MOV   R2,#5
CMP   R1,R2
```

R0	0x0	Dec	Bin	Hex
R1	0b101	Dec	Bin	Hex
R2	0b101	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x10	Dec	Bin	Hex

Clock Cycles Current Instruction: 1 Total: 3

CSPR Status Bits (NZCV)	0	1	1	0

```
MOV   R1,#5
MOV   R2,#2
CMP   R1,R2
```

R0	0x0	Dec	Bin	Hex
R1	0b101	Dec	Bin	Hex
R2	0b10	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x10	Dec	Bin	Hex

Clock Cycles Current Instruction: 1 Total: 3

CSPR Status Bits (NZCV)	0	0	1	0

```
MOV   R1,#5
MOV   R2,#7
CMP   R1,R2
```

R0	0x0	Dec	Bin	Hex
R1	0x5	Dec	Bin	Hex
R2	0x7	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x10	Dec	Bin	Hex

Clock Cycles Current Instruction: 1 Total: 3

CSPR Status Bits (NZCV)	1	0	0	0

CMP, TST, and TEQ

- **Setting condition code flags**

  CMP, TST, and TEQ, always update the condition code flags

  Arithmetic, Logic, and Move instructions do so only if S is appended to the OP code

  ADDS updates flags, but ADD does not

Instruction set

- Data processing
- Data movement (memory access)
- Flow control

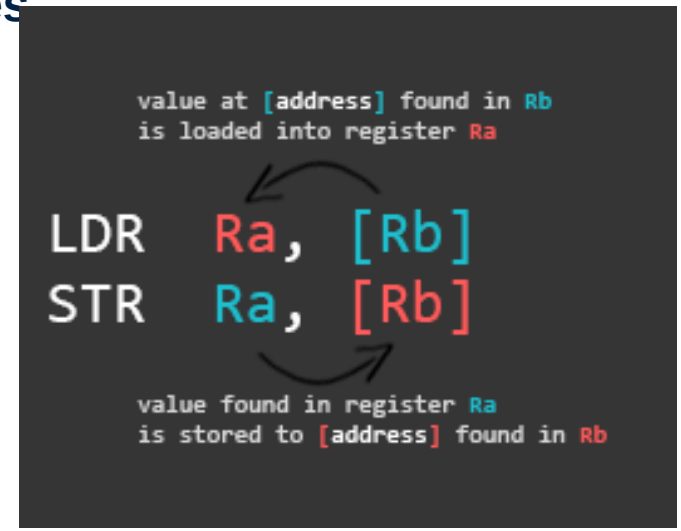
Single register data transfer

LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- **Memory system must support all access sizes**

- **Syntax:**
  - LDR{<cond>}{<size>} Rd, <address>
  - STR{<cond>}{<size>} Rd, <address>

  e.g. LDREQB



Address accessed

- **Address accessed by LDR/STR is specified by a base register plus an offset**
- **For word and unsigned byte accesses, offset can be**
  - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).

    ```
    LDR r0,[r1,#8]
    ```
  - A register, optionally shifted by an immediate value

    ```
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
    ```
- **This can be either added or subtracted from the base register:**

  ```
  LDR r0,[r1,#-8]
  LDR r0,[r1,-r2]
  LDR r0,[r1,-r2,LSL#2]
  ```
- **For halfword and signed halfword / byte, offset can be:**
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- **Choice of *pre-indexed* or *post-indexed* addressing**

Index Method	Data	Base Address Register	Example
preindex with writeback	mem[base+offset]	base+ offset	LDR r0, [r1, #4]!
preindex	mem[base+offset]	not updated	LDR r0, [r1, #4]
postindex	mem[base]	base+ offset	LDR r0, [r1], #4

Comparisons

- Pre-indexed addressing

```
LDR   R0, [R1, R2]    @ R0=mem[R1+R2]
                      @ R1 unchanged
```

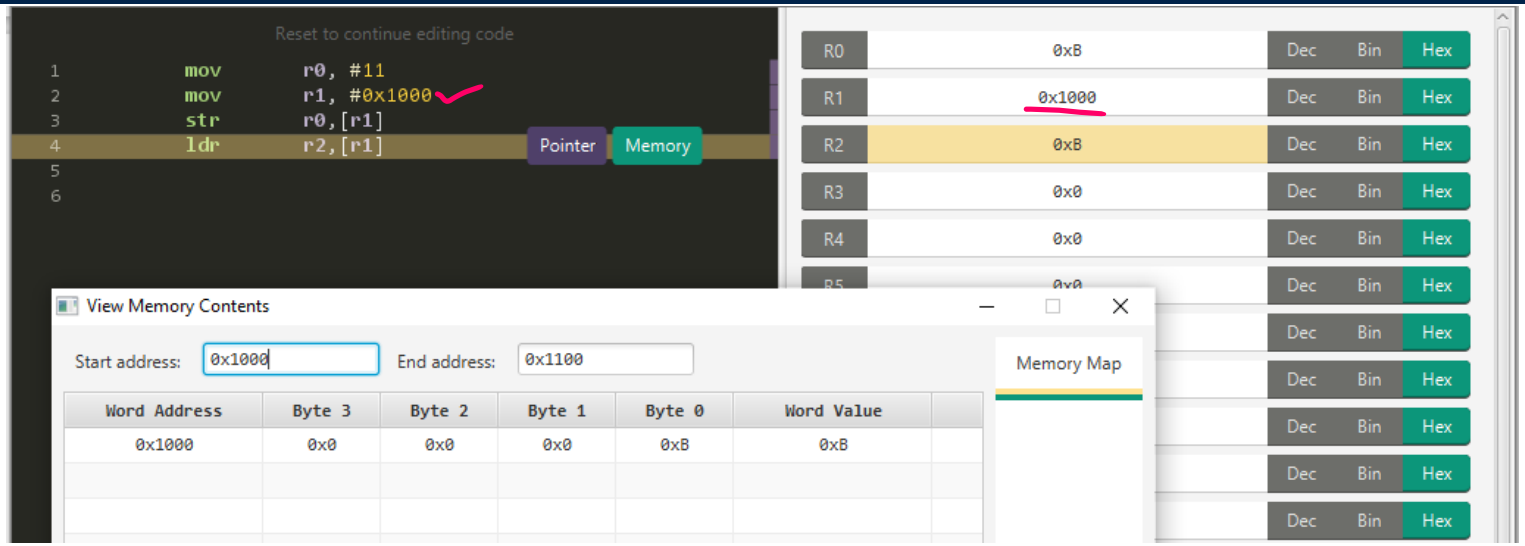
- Auto-indexing addressing Or Pre-indexed with writeback

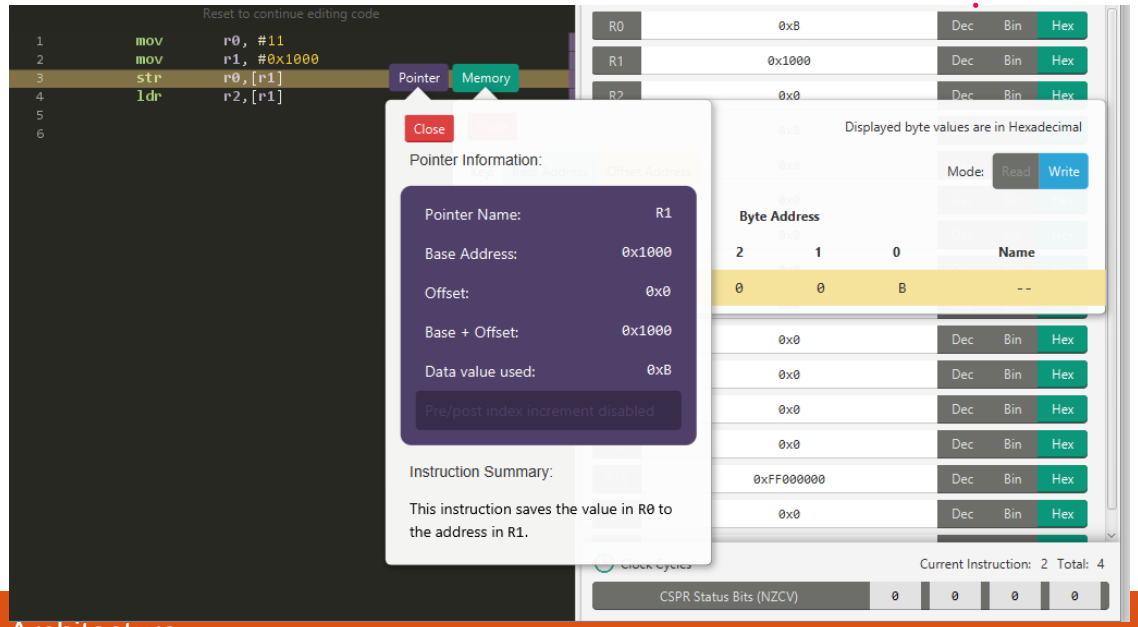
```
LDR   R0, [R1, R2]!   @ R0=mem[R1+R2]
                      @ R1=R1+R2
```

- Post-indexed addressing

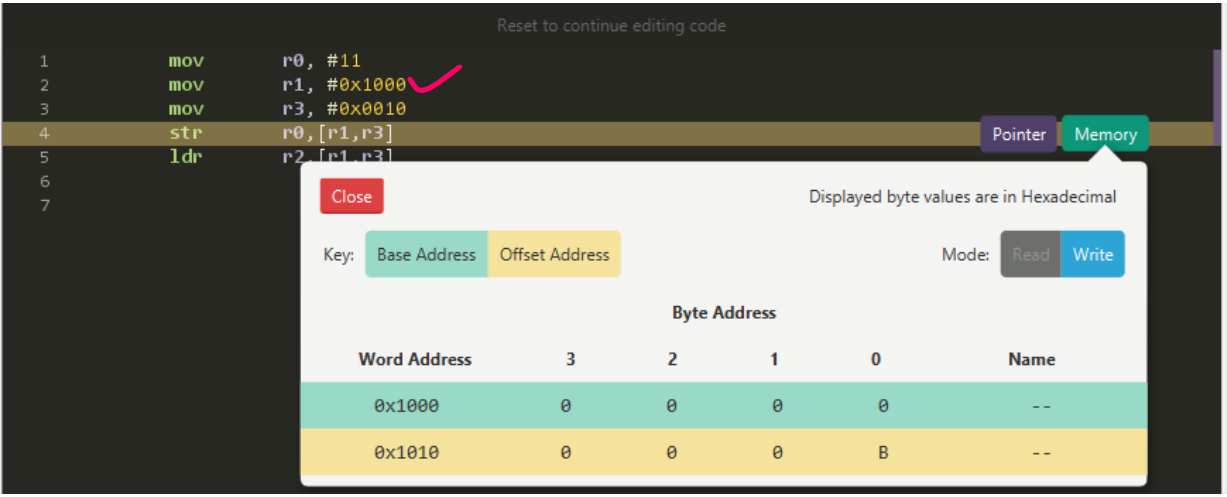
```
LDR   R0, [R1], R2    @ R0=mem[R1]
                      @ R1=R1+R2
```

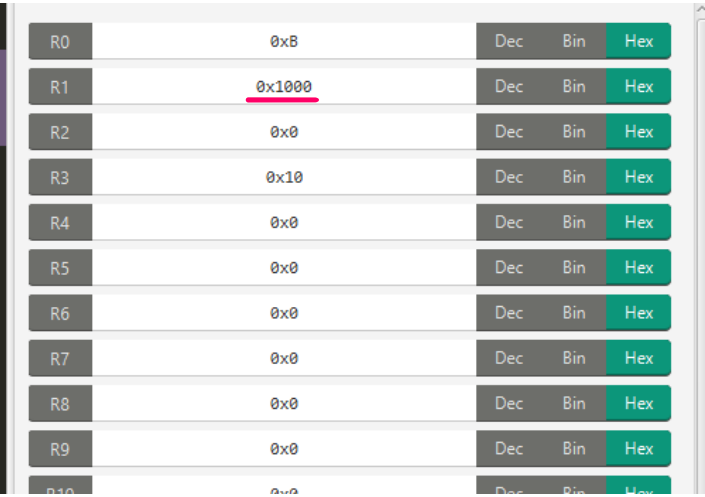
Pre-indexed Addressing

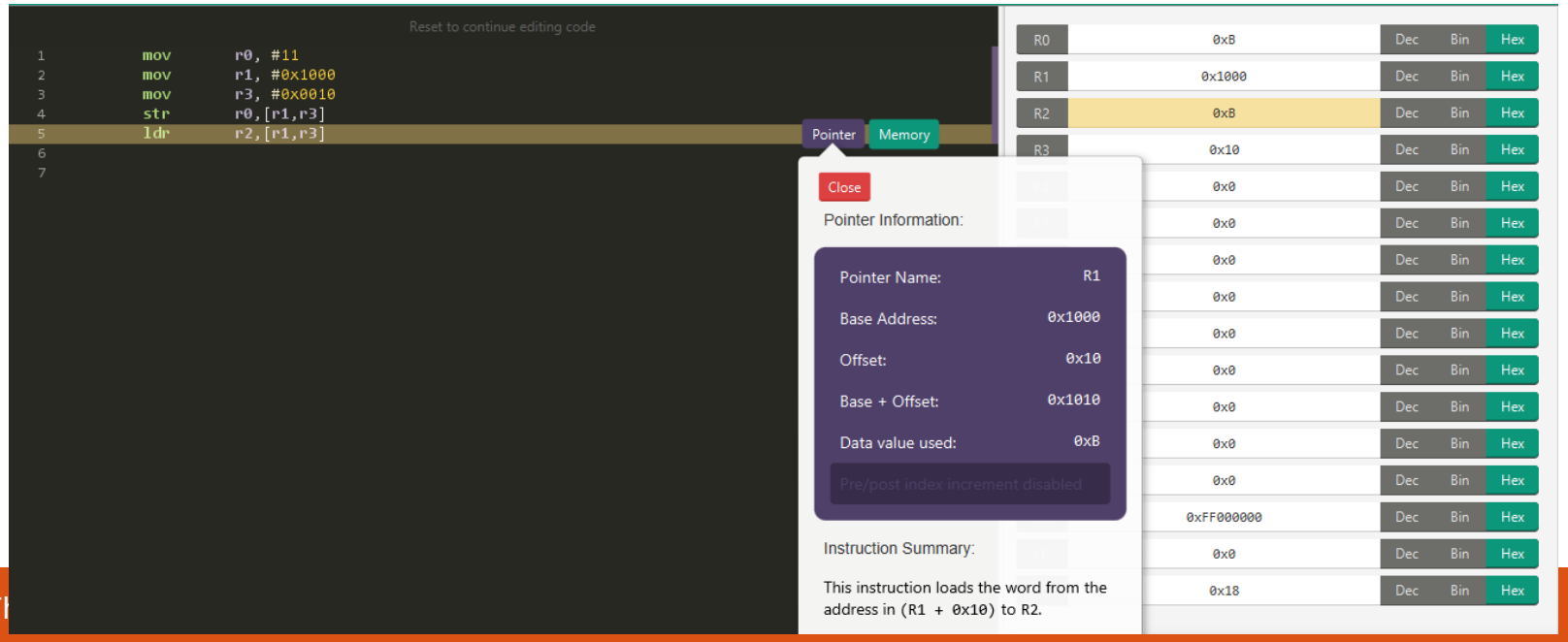




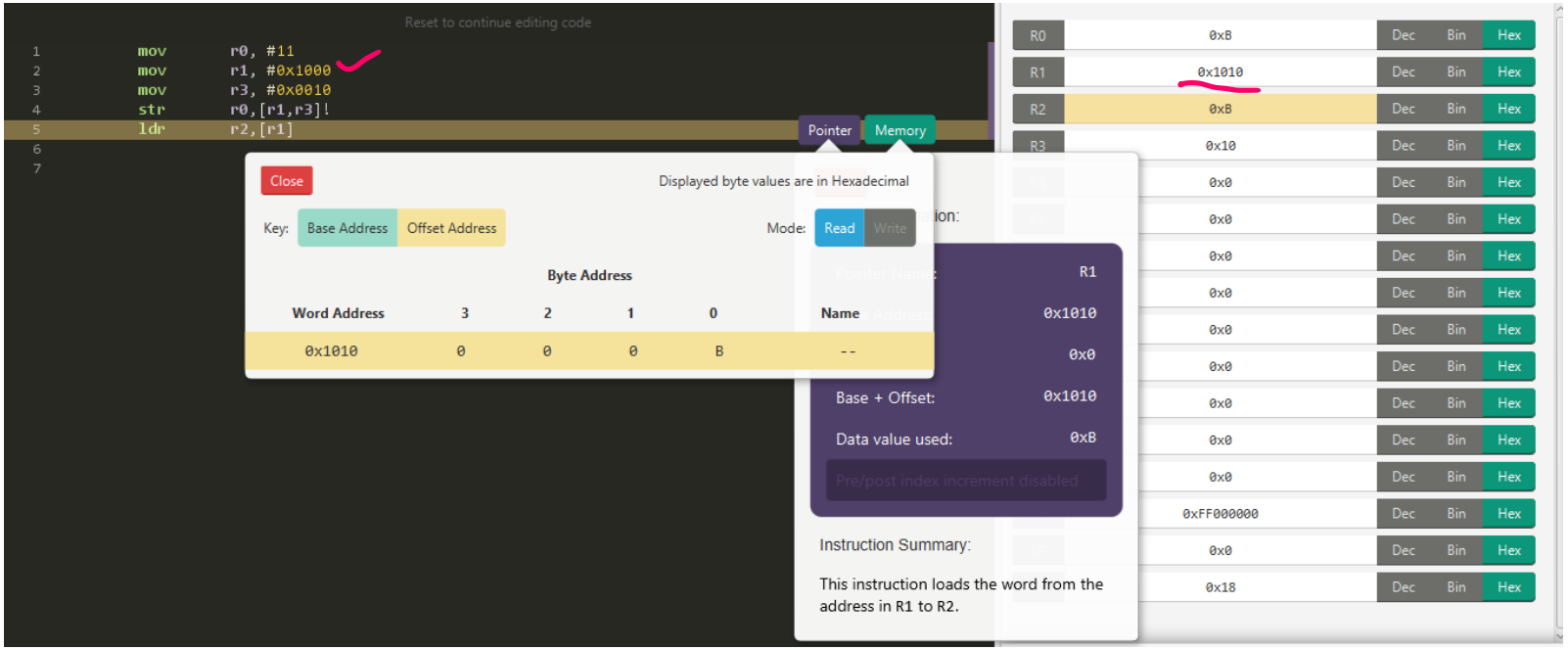
Pre-indexed Addressing

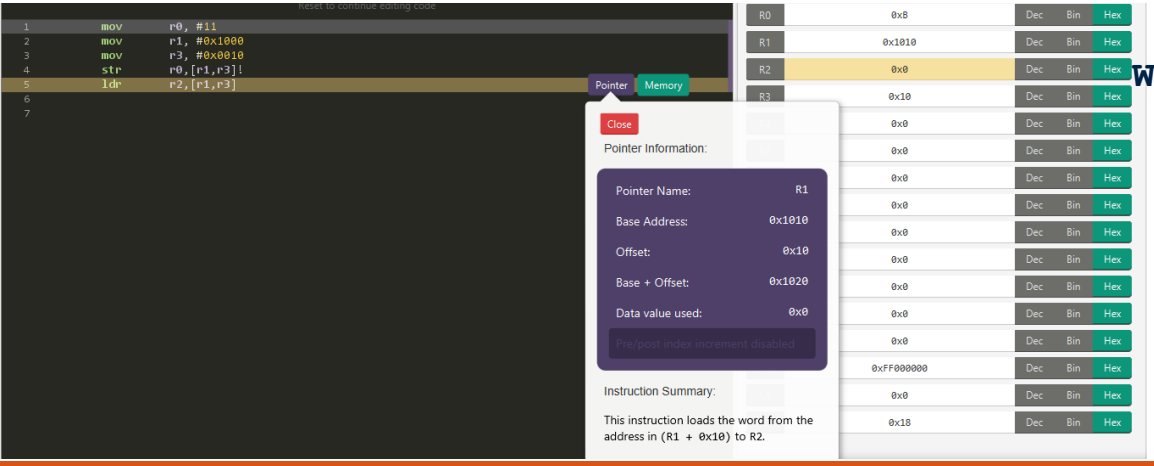






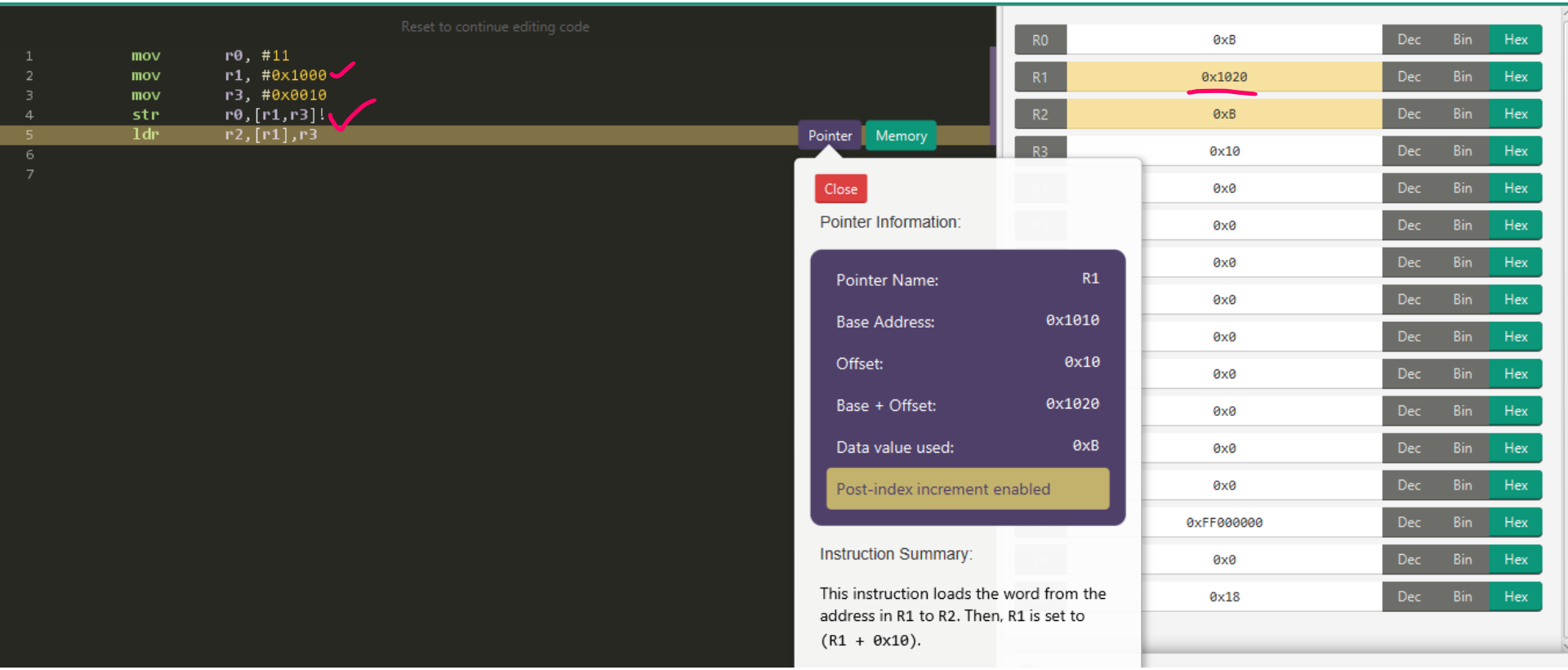
Auto-indexed Addressing





Why R2 contains 0x0, and not 0xB?

Post-indexed Addressing



```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)   r0 = 0x02020202
          r1 = 0x00009004
```

```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1, #4]
```

Preindexing:

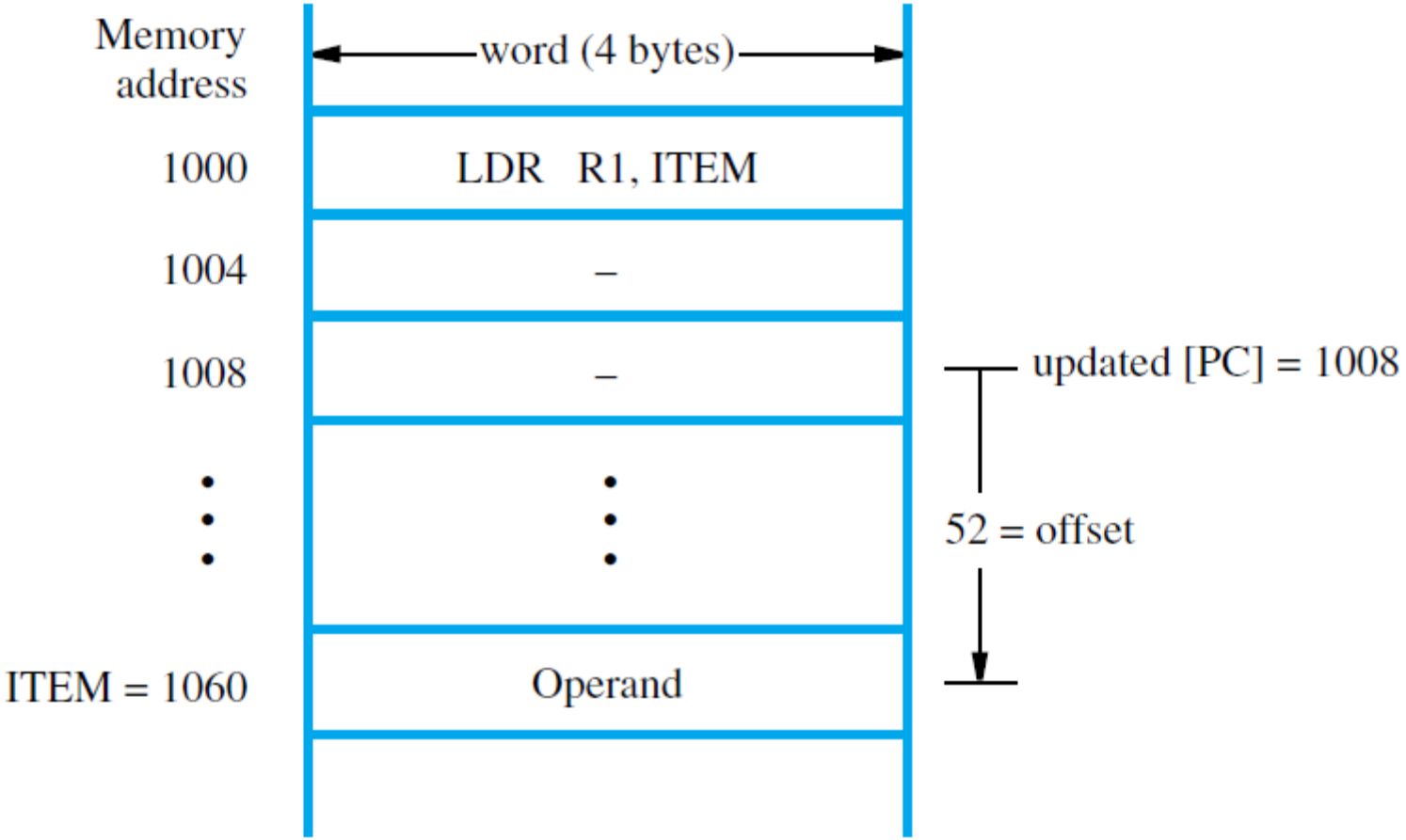
```
POST(2)   r0 = 0x02020202
          r1 = 0x00009000
```

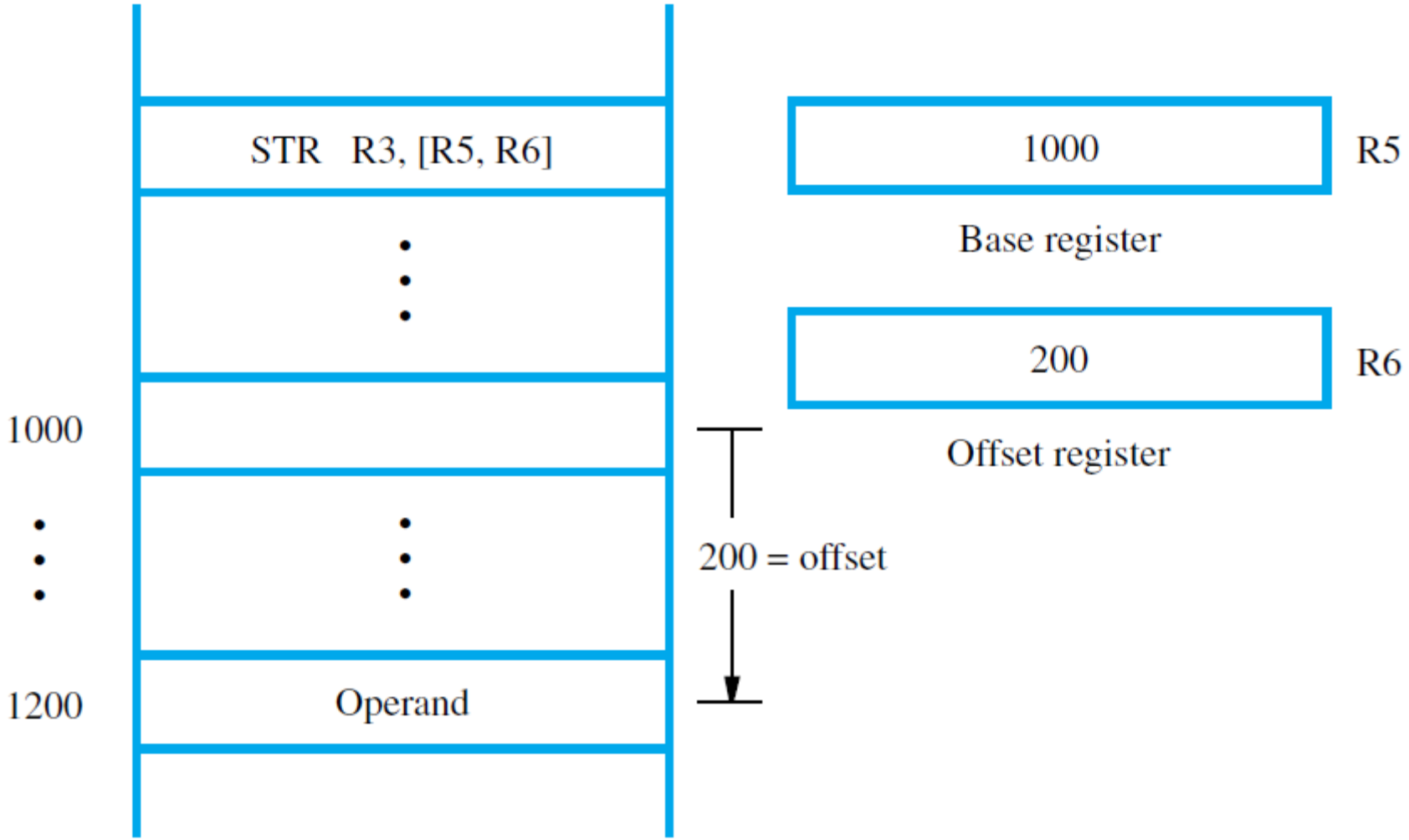
```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1], #4
```

Postindexing:

```
POST(3)   r0 = 0x01010101
          r1 = 0x00009004
```


Memory address
word (4 bytes)
1000
LDR R1, ITEM
1004
–
1008
–
updated [PC] = 1008
52 = offset
ITEM = 1060
Operand


STR R3, [R5, R6]
1000
1200
Operand
200 = offset
1000
R5
Base register
200
R6
Offset register

LDM / STM operation: Block Transfer

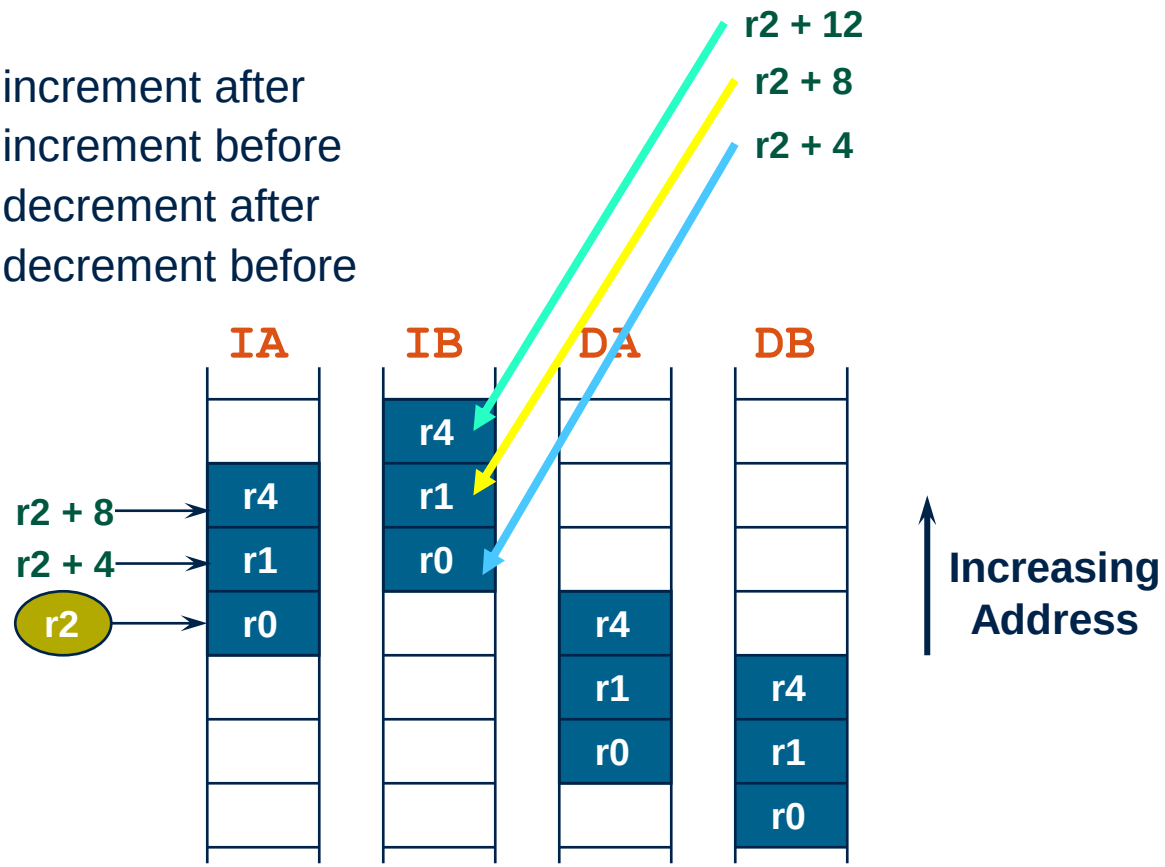
- **Syntax:**

  `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>

- **4 addressing modes:**

`LDMIA / STMIA`	increment after
`LDMIB / STMIB`	increment before
`LDMDA / STMDA`	decrement after
`LDMDB / STMDB`	decrement before

```
LDMxx r2, {r0,r1,r4}
STMxx r2, {r0,r1,r4}
```

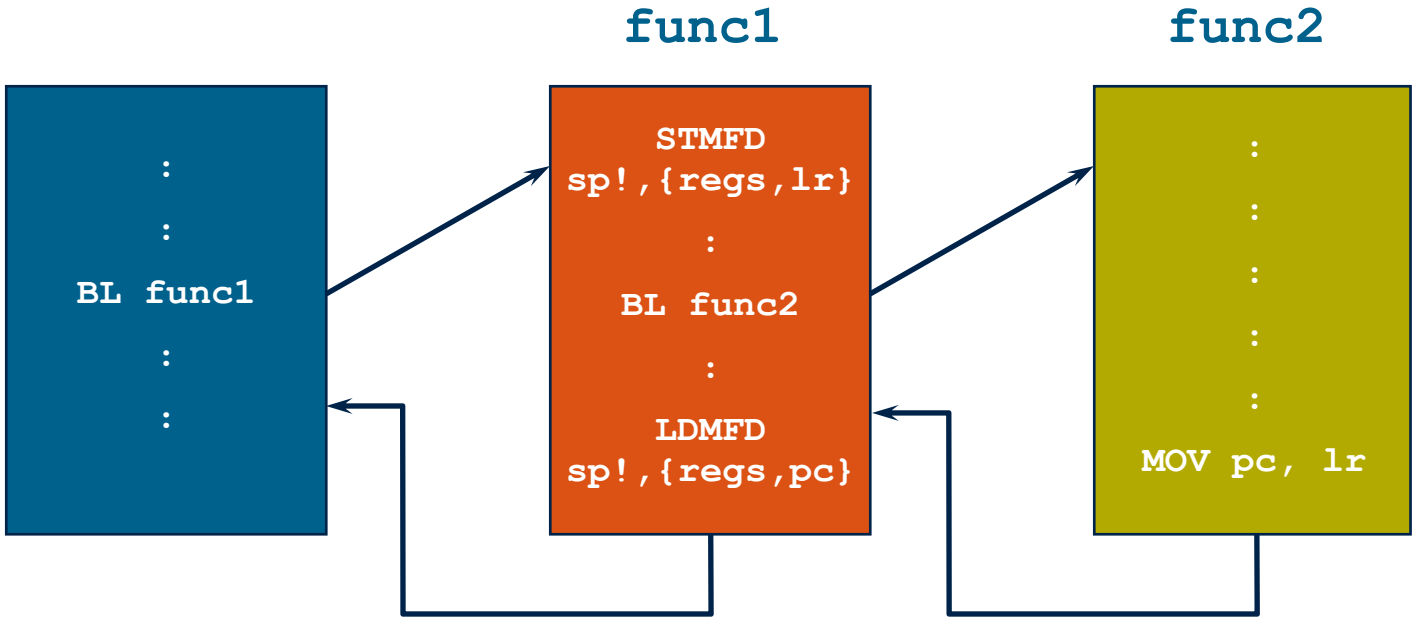


Instruction set

- Data processing
- Data movement (memory access)
- Flow control

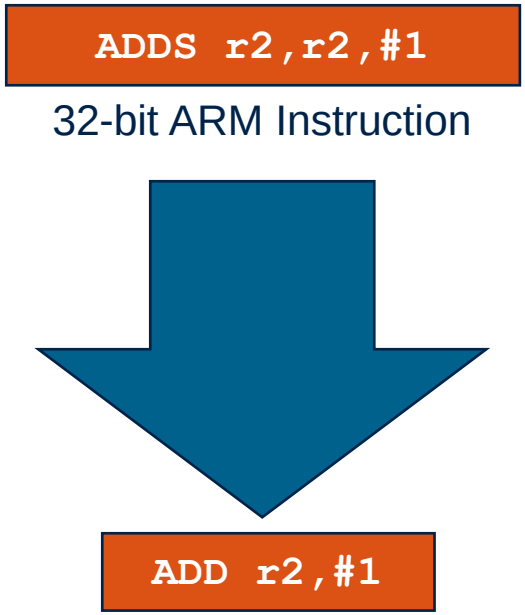
ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



Thumb

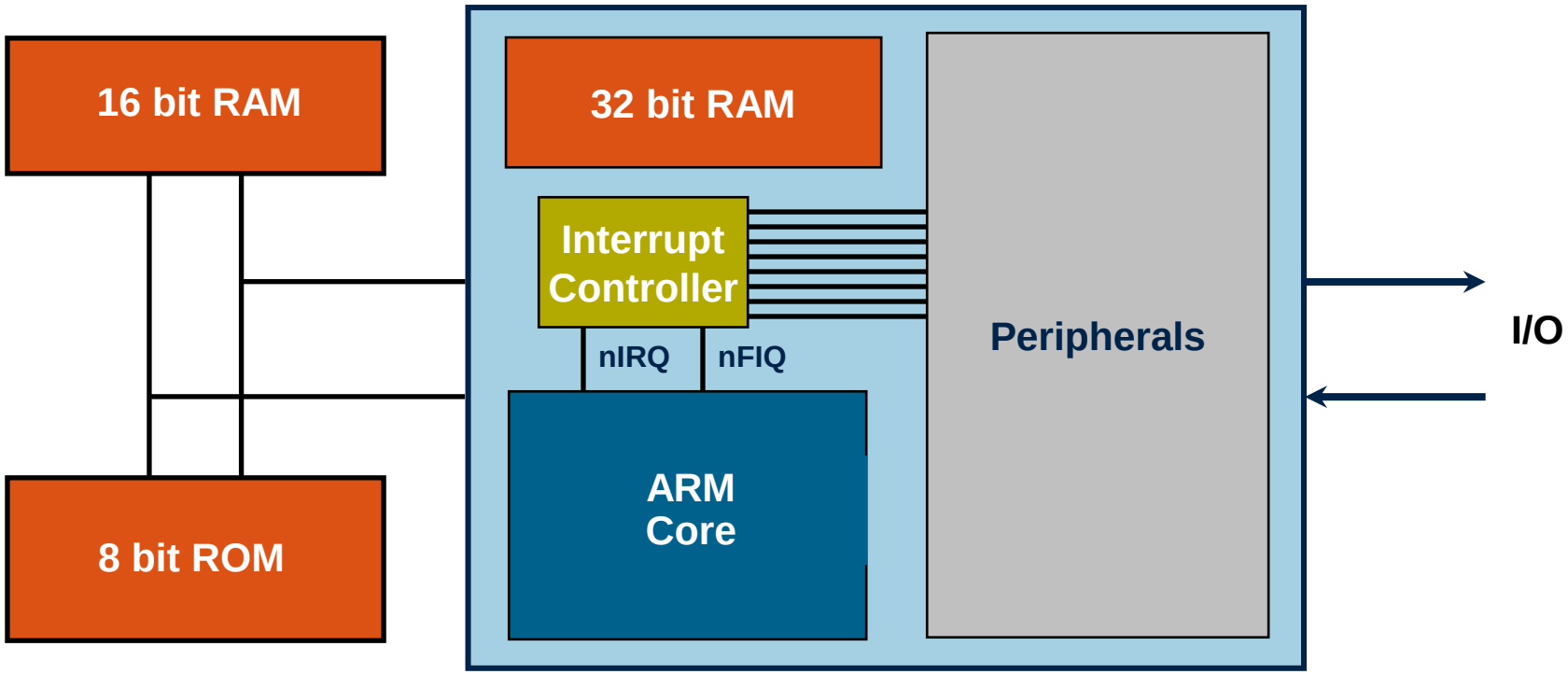
- **Thumb is a 16-bit instruction set**
  - Optimised for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- **Core has additional execution state - Thumb**
  - Switch between ARM and Thumb using `BX` instruction



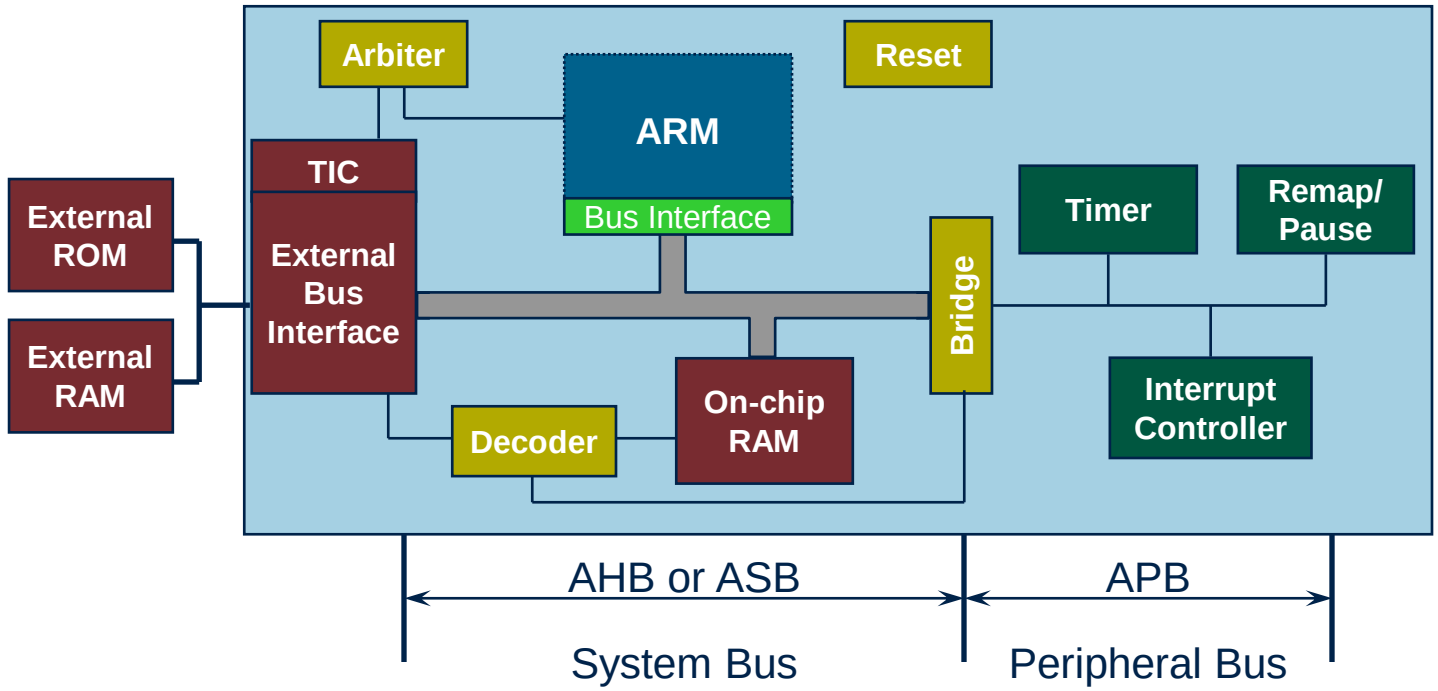
For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Example ARM-based System



AMBA (ARM Advanced Microcontroller Bus Architecture



- **AMBA**
  - Advanced Microcontroller Bus Architecture
- **ADK**
  - Complete AMBA Design Kit
- **ACT**
  - AMBA Compliance Testbench
- **PrimeCell**
  - ARM's AMBA compliant peripherals

Condition Codes

- **The possible condition codes are listed below:**
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

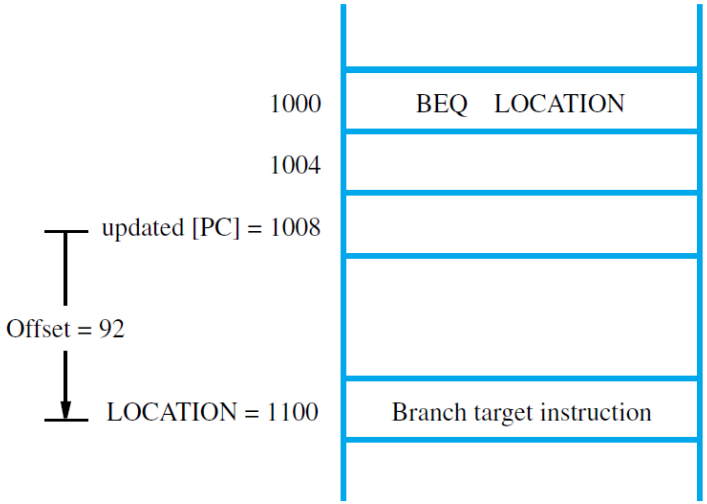
Condition Codes

- **Branch:**

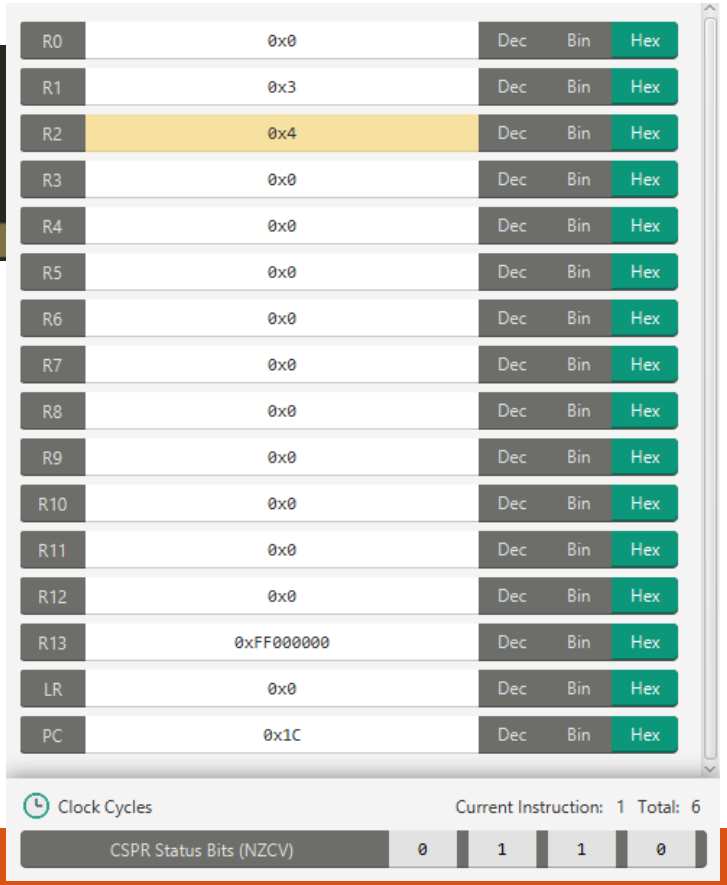
  B{condition} LOCATION

  branches to LOCATION if the settings of the condition code flags satisfy {condition}

  BEQ LOCATION
  branches if Z = 1




```
1           MOV     R1, #3
2           CMP     R1,#3
3           BEQ     EQUAL
4           ADD     R2,R1,#5
5           END
6 EQUAL     ADD     R2,R1,#1
```



ARM

- Unconditional jump

```
        B     LABEL
        …
LABEL   …
```

- Loop ten times

```
        MOV r0,#10
Loop        …
        SUBS r0,#1
        BNE Loop
        …
```

- Call a subroutine

```
        BL SUB
        …
SUB         …
        MOV PC,r14
```

- Conditional subroutine call

```
    CMP r0,#5
    BLLT SUB1 ;if r0<5,
              ;call sub1
    BLGE SUB2 ;else call
              ;SUB2
```

Conditional Execution and Flags

- **ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.**
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
 CMP   r3,#0
  BEQ   skip
  ADD   r0,r1,r2
skip
```

```
CMP   r3,#0
 ADDNE r0,r1,r2
```

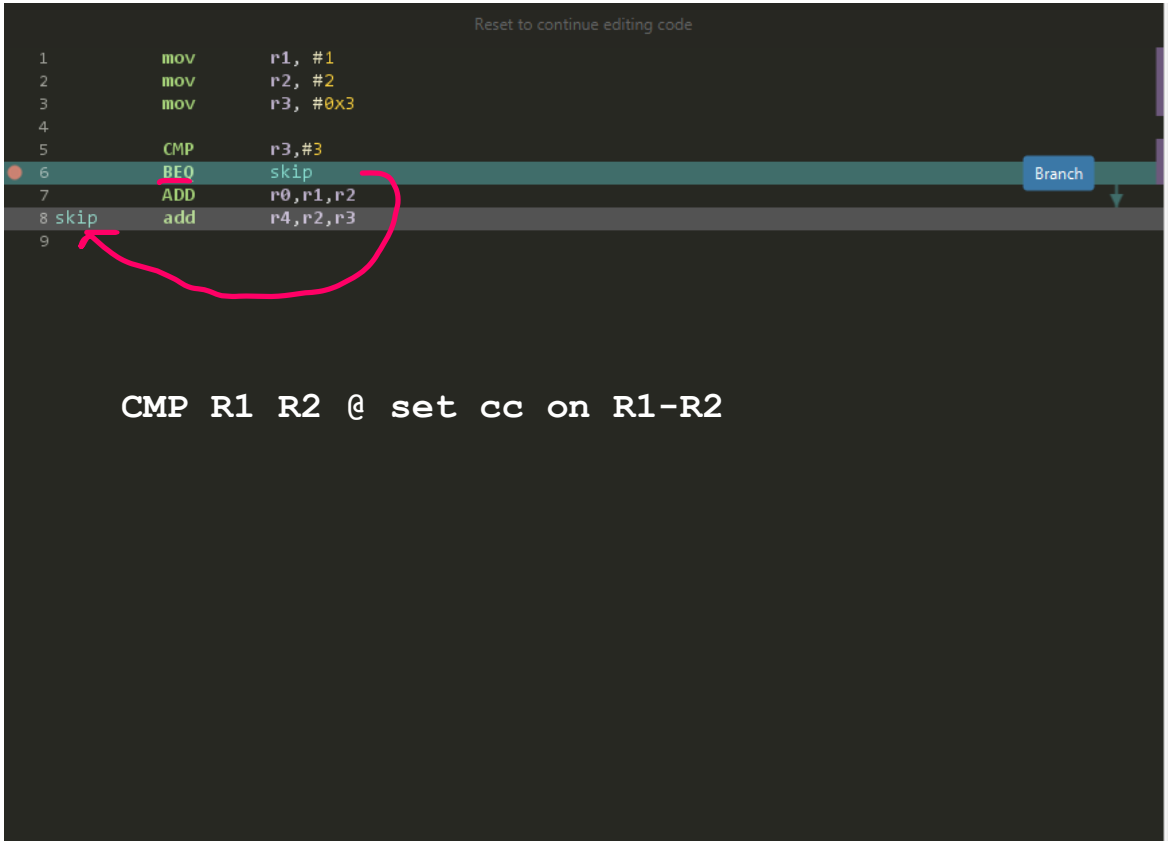
- **By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".**

```
loop
  …
  SUBS r1,r1,#1
  BNE loop
```

decrement r1 and set flags

if Z flag clear then branch

ARM


Reset to continue editing code
1 mov r1, #1
2 mov r2, #2
3 mov r3, #0x3
4
5 CMP r3,#3
6 BEQ skip
7 ADD r0,r1,r2
8 skip add r4,r2,r3
9
Branch
CMP R1 R2 @ set cc on R1-R2


R0 0x0 Dec Bin Hex
R1 0x1 Dec Bin Hex
R2 0x2 Dec Bin Hex
R3 0x3 Dec Bin Hex
R4 0x0 Dec Bin Hex
R5 0x0 Dec Bin Hex
R6 0x0 Dec Bin Hex
R7 0x0 Dec Bin Hex
R8 0x0 Dec Bin Hex
R9 0x0 Dec Bin Hex
R10 0x0 Dec Bin Hex
R11 0x0 Dec Bin Hex
R12 0x0 Dec Bin Hex
R13 0xFF000000 Dec Bin Hex
LR 0x0 Dec Bin Hex
PC 0x18 Dec Bin Hex
Clock Cycles
Current Instruction: 3 Total: 7
CSPR Status Bits (NZCV) 0 1 1 0

Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP     r0,#0
    MOVEQ   r0,#1
    BLEQ    func
```

- Set the flags, then use various condition codes

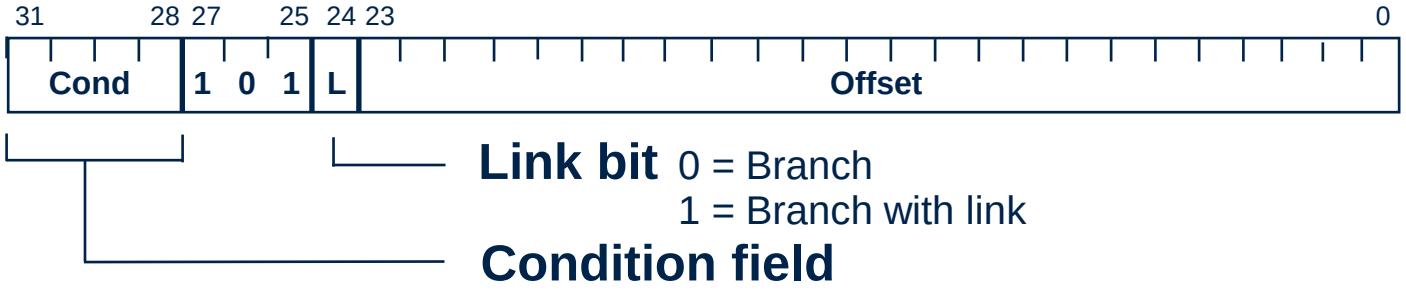
```
if (a==0) x=0;
if (a>0)  x=1;
    CMP     r0,#0
    MOVEQ   r1,#0
    MOVGT   r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP     r0,#4
    CMPNE   r0,#10
    MOVEQ   r1,#0
```

Branch instructions

- **Branch :** `B{<cond>} label`
- **Branch with Link :** `BL{<cond>} subroutine_label`



- **Subroutine linkage:**

  BL SUBADDRESS

  Actions taken:

  1. **The value of the updated PC is stored in R14 (LR), the Link register**
  2. **A branch is taken to SUBADDRESS**

```
 9                MOV       R0, #15
10                MOV       R1, #20
11                BL        FUNCTION_1
12                MOV       R3,#88
13
14 FUNCTION_1     ADD       R2,R0,R1
15                MOV       PC,LR
16
```

Branch

```
 9                MOV       R0, #15
10                MOV       R1, #20
11                BL        FUNCTION_3
12                MOV       R3,#88
13
14 FUNCTION_3     ADD       R2,R0,R1
15                ;MOV      PC,LR  WHAT HAPPEN IF WE COMMENT THIS INSTRUCTION?
```

```
18                MOV       R0, #15
19                MOV       R1, #20
20                B         FUNCTION_2
21                MOV       R3,#88
22
23 FUNCTION_2     ADD       R2,R0,R1
24                MOV       PC,LR
```

Branch

```
 9                MOV       R0, #15
10                MOV       R1, #20
11                B         FUNCTION_4
12                MOV       R3,#88
13
14 FUNCTION_4     ADD       R2,R0,R1
15                ;MOV      PC,LR  WHAT HAPPEN IF WE COMMENT THIS INSTRUCTION?
```


ARM®
THE ARCHITECTURE
FOR THE DIGITAL WORLD™