08 Addressing Modes

CSC 230

Stallings: Ch 12, 13; Appendix B.1
M&H: 4.2 to 4.6 (translated from ARC)

INSTRUCTION FORMATS and DESIGN

Instructions always contains:

(1) opcode

(2) one of more bytes for operands

- ❑ **Mnemonics are used for opcodes (symbolic names)**
- ❑ **Registers (or other) are used for operands**
- ❑ **Note that the number of bits for the opcode (operation code) subdivision must be large enough to be able to represent *all* instructions in instruction set**

3 OPERANDS FORMAT

opcode	F1 (source)	F2 (source)	F3 (destination)

or

opcode	F3 (destination)	F2 (source)	F3 (source)

❑ Operation (Opcode) applied on source operands

❑ Result in destination operand

❑ F1, F2 and F3 are memory addresses or registers

→ length?

```
ADD  d, s1, s2          @ d = s1 + s2
```

(Most ARM instructions, with registers only as operands)

2 OPERANDS FORMAT

opcode	F1 (source)	F2 (destination)

or

opcode	F1 (destination)	F2 (source)

- ❑ one of F1 or F2 is usually a register
- ❑ Operation applied on both F1 and F2
- ❑ result in one of them

→ length

```
ADD  d, s1      @ d = d + s1
```

1 OPERAND FORMAT

opcode F1

- usually there is an implicit register or accumulator or constant

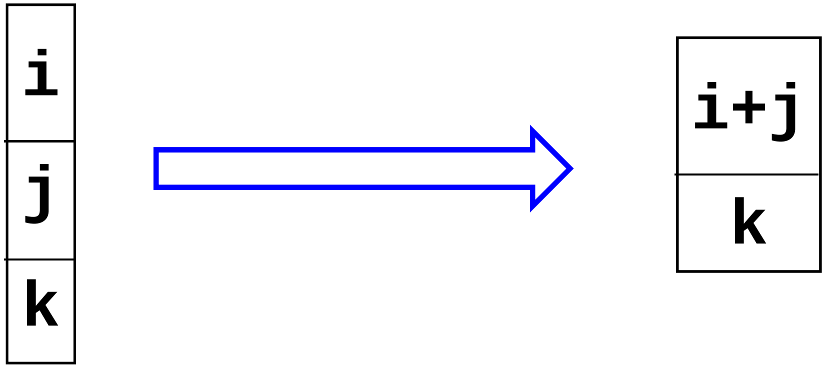
```
INCR s1    @ s1 = s1 + 4
```

1 OPERAND FORMAT or STACK FORMAT

opcode

- need instruction only, assumes there is a stack of operands
- takes top element of the stacks by pop and performs operation
- result is stored by pushing on stack again

```
ADD   @top-of-stack = top-of-stack + next-on-stack
```



One, Two, Three Operands Machines

Consider how the C expression

A = B*C + D

might be evaluated by each of the one, two, and three-address instruction types.

❑ Assumptions for this example: Addresses and data words are two bytes in size. Opcodes are 1 byte in size. Operands are moved to and from memory one word (two bytes) at a time.

a) Three Operands Machines

A = B*C + D

In a three-address instruction, the expression A = B*C + D might be coded as:

```
mult B, C, A    // A = B * C
add  D, A, A    // A = A + D
```

which means:

i. multiply B by C and store the result at A.

ii. Then, add D to A and store the result at address A.

b) Two Operands Machines

A = B*C + D

In a two-address instruction, one of the operands is overwritten by the result. Here, the code for the expression A = B*C + D is:

```
load B, A        // A = B
mult C, A        // A = A * C
add  D, A        // A = A + D
```

c) One Operand (Accumulator) Machines

A = B*C + D

A one-address instruction employs a single arithmetic register in the CPU, known as the accumulator. The code for the expression A = B*C + D is now:

```
load  B
mult  C
add   D
store A
```

A. The load instruction loads B into the accumulator,
B. mult multiplies C by the accumulator and stores the result in the accumulator,
C. add does the corresponding addition.
D. The store instruction stores the accumulator in A.

d) Zero Operand (Stack) Machines

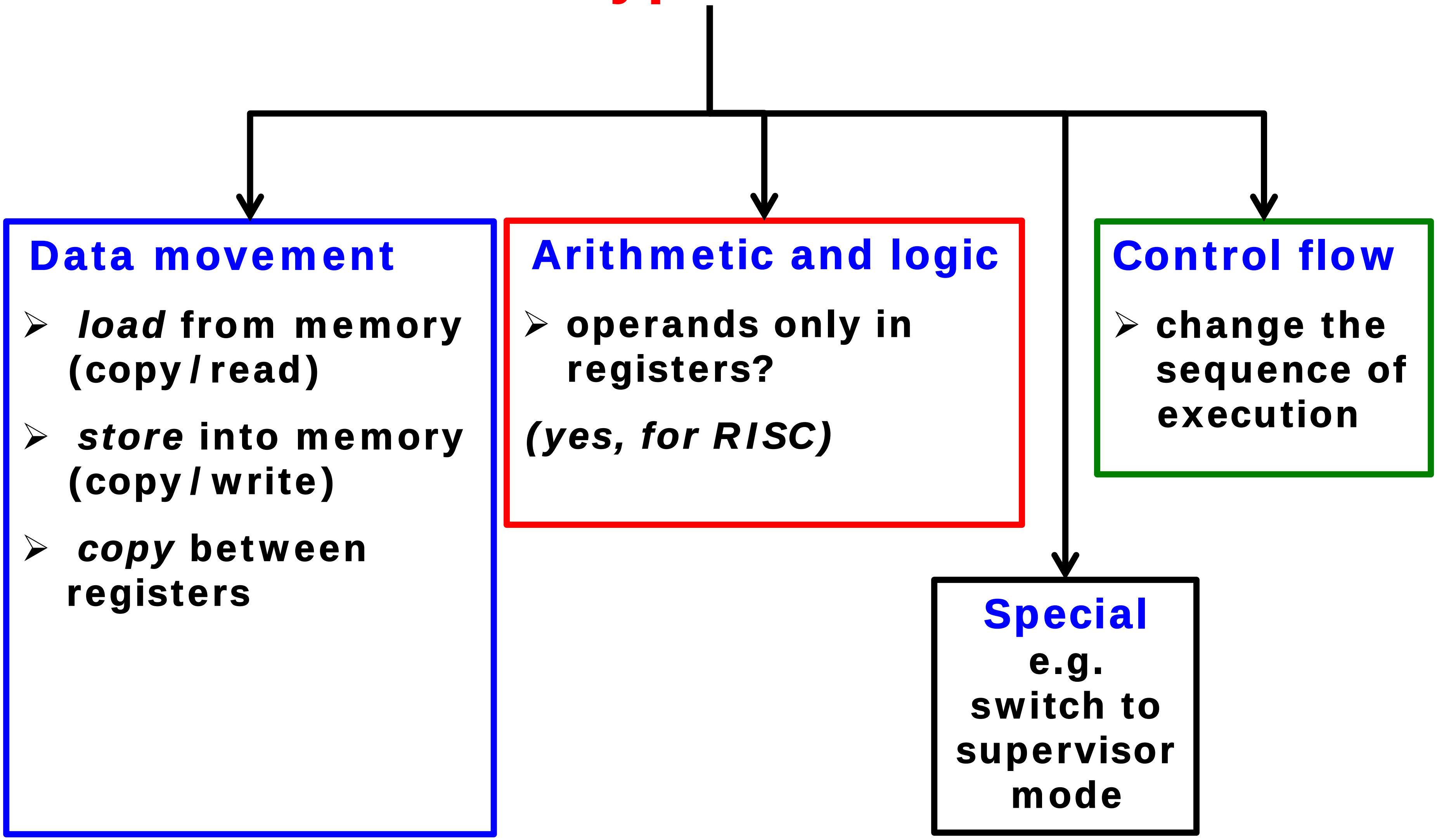
A = B*C + D

All operands are obtained from a stack.
The results of instructions are pushed back onto the stack.
The code for the expression A = B*C + D would be:

```
load B     # push copy of B onto stack
load C     # push copy of C onto stack
mult       # pop twice, multiply, push result
load D     # push copy of D onto stack
add        # pop twice, add, push result
storeA     # pop, store into A
```

- Better code density but, if the stack is held in main memory, there is a lot more memory traffic when executed.
- JVM and the .NET runtime (aka CLR) are examples of zero-address architectures, but not implemented in hardware.

Instruction Types and execution



Data movement

- *load* from memory (copy / read)
- *store* into memory (copy / write)
- *copy* between registers

Arithmetic and logic

- operands only in registers?

(yes, for RISC)

Control flow

- change the sequence of execution

Special

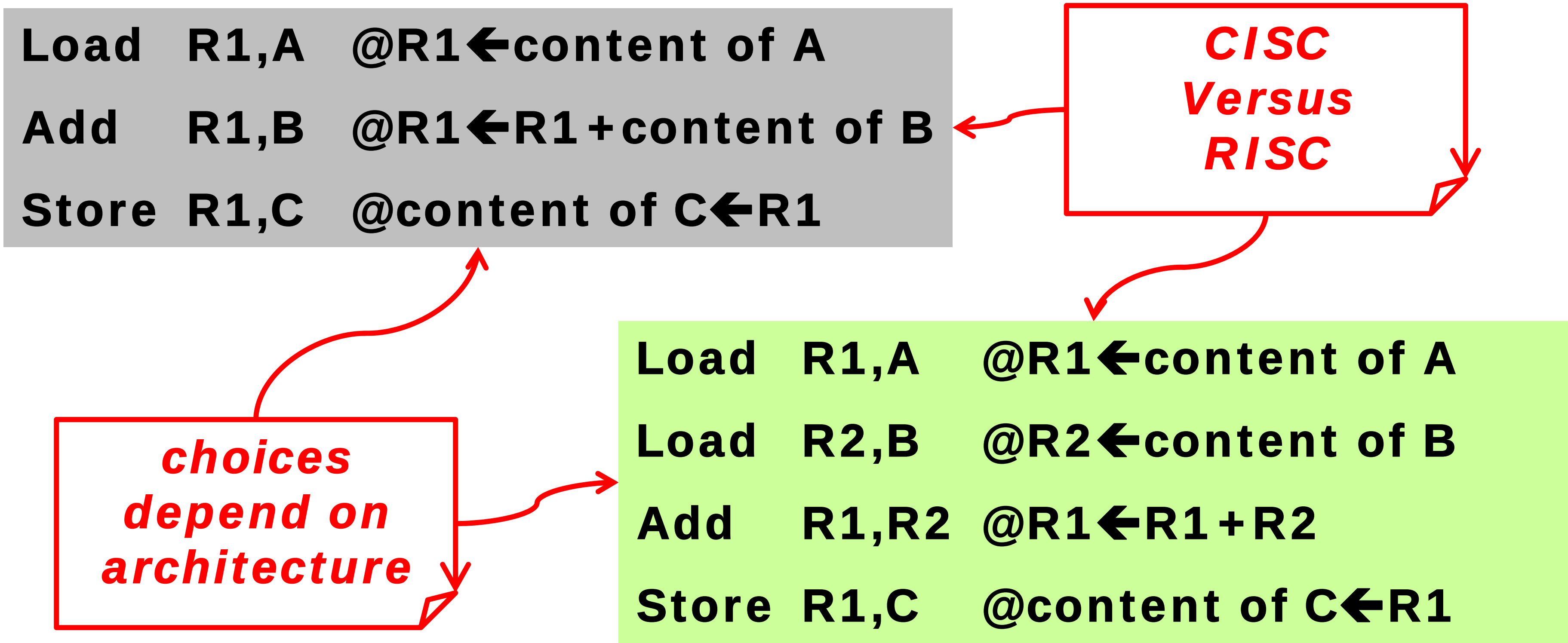
e.g.
switch to
supervisor
mode

See also Stallings Table 12.3

From High level to Low level

C = A + B   in a high level language

NOTE: A, B, C denote memory locations with some content in them – `int A,B,C;`



```
Load   R1,A   @R1←content of A
Add    R1,B   @R1←R1+content of B
Store  R1,C   @content of C←R1
```

CISC Versus RISC

```
Load   R1,A    @R1←content of A
Load   R2,B    @R2←content of B
Add    R1,R2   @R1←R1+R2
Store  R1,C    @content of C←R1
```

choices depend on architecture

From High level to Low level: to further details

C = A + B   in a high level language

```
Load   R1,A   @R1←content of A
Add    R1,B   @R1←R1+content of B
Store  R1,C   @content of C←R1
```

```
Load   R1,A    @R1←content of A
Load   R2,B    @R2←content of B
Add    R1,R2   @R1←R1+R2
Store  R1,C    @content of C←R1
```

```
LoadAddr    R3,A        @R3←address of A
Load        R1, [R3]    @R1←content of A, indexed by R3
LoadAddr    R4,B        @R4←address of B
Load        R2,[R4]     @R2←content of B, indexed by R4
Add         R1,R2       @R1←R1+R2
LoadAddr    R5,C        @R5←address of C
Store       R1,[R5]     @content of C←R1, indexed by R5
```

From High level to Low level: to ARM

C = A + B   in a high level language

```
Load    R1,A    @R1←content of A
Add     R1,B    @R1←R1+content of B
Store   R1,C    @content of C←R1
```

```
Load    R1,A    @R1←content of A
Load    R2,B    @R2←content of B
Add     R1,R2   @R1←R1+R2
Store   R1,C    @content of C←R1
```

```
LoadAddrR3,A       @R3←address of A         LDR   R3,=A
Load    R1, [R3]   @R1←content of A,        LDR   R1,[R3]
LoadAddrR4,B       @R4←address of B         LDR   R4,=B
Load    R2,[R4]    @R2←content of B         LDR   R2,[R4]
Add     R1,R2      @R1←R1+R2                ADD   R1,R1,R2
LoadAddrR5,C       @R5←address of C         LDR   R5,=C
Store   R1,[R5]    @content of C←R1         STR   R1,[R5]
```

Instruction Set and ISA

Instruction set = the collection of all instructions that a processor can execute

ISA = Instruction Set Architecture = collection of instruction set, memory, accessible registers

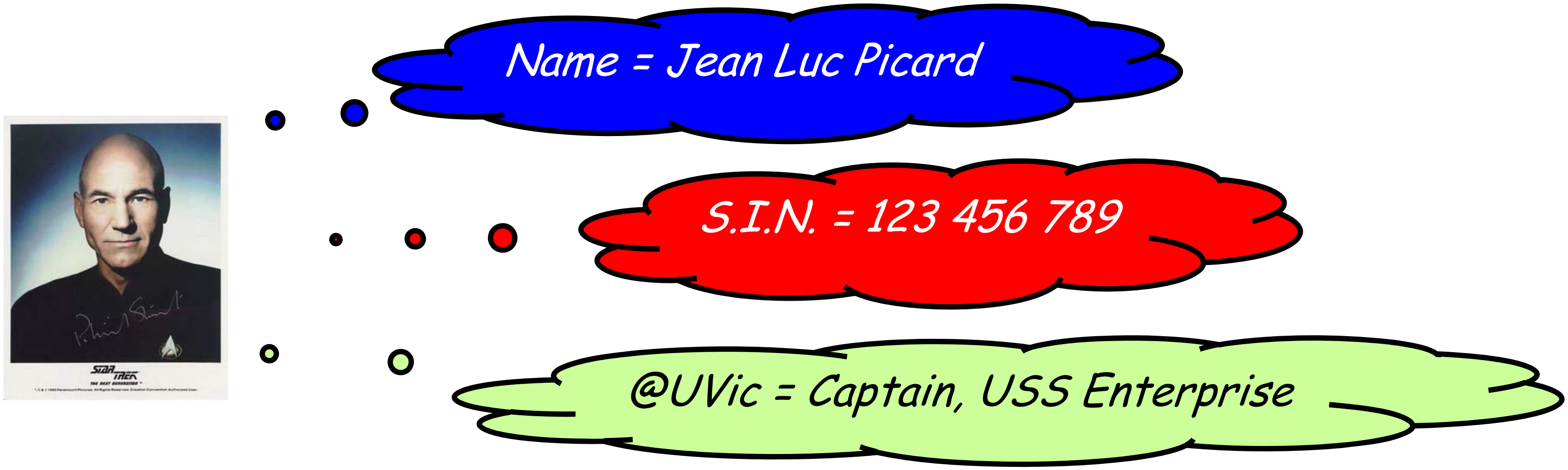
Instruction Class	C	VAX Assembly
Data movement	a = b	MOV b, a
Arithmetic and logic	b = c + d * e	MPY d, e, b ADD c, b, b
Control flow	goto Label	BR LABEL
I/O	printf	never in ISA

Addressing Modes

Addressing modes are an aspect of the ISA

They define how instructions identify the operands by specifying how to calculate their effective memory address.
They achieve this by using information held in registers and/or constants contained within a machine instruction (or elsewhere, e.g. extension words in 68000).

(paraphrased from Wikipedia)



Name = Jean Luc Picard

S.I.N. = 123 456 789

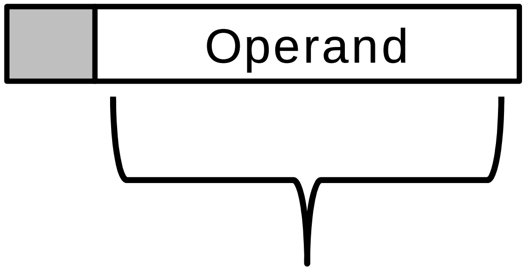
@UVic = Captain, USS Enterprise

Addressing modes: a very GENERAL introduction.1

Immediate addressing

instruction contains the value of the operand

```
move R1, #3
```



limitations on size

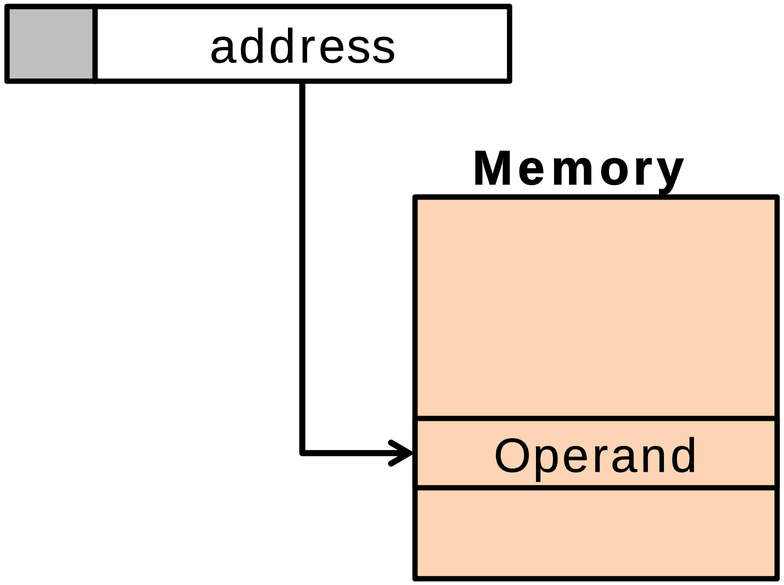
Addressing modes: a very GENERAL introduction.2

Direct addressing

Absolute addressing

instruction contains the binary address of operand (short)

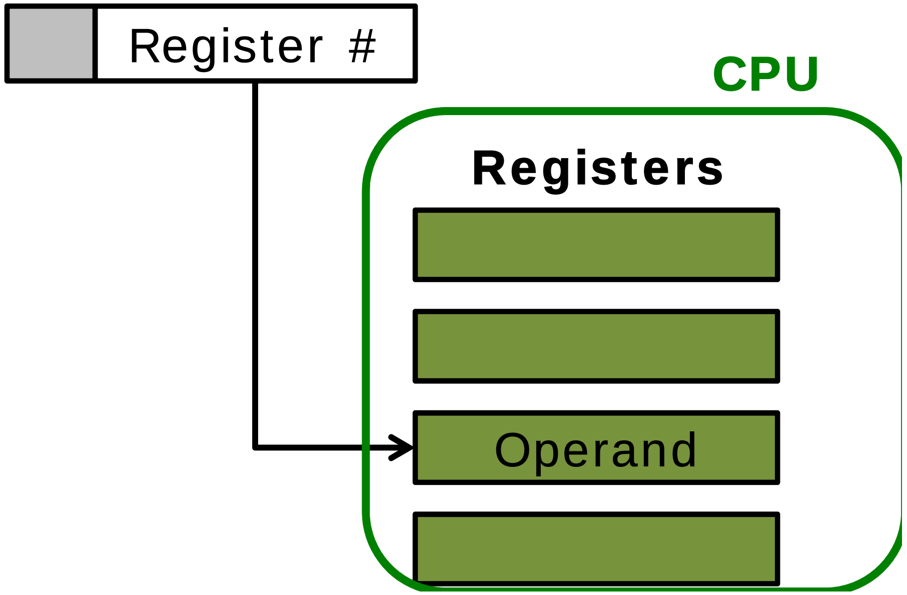
load R1, 0xC000



Register Direct addressing

- instruction contains a register number
- operand is the content of the register

Add R1, R2



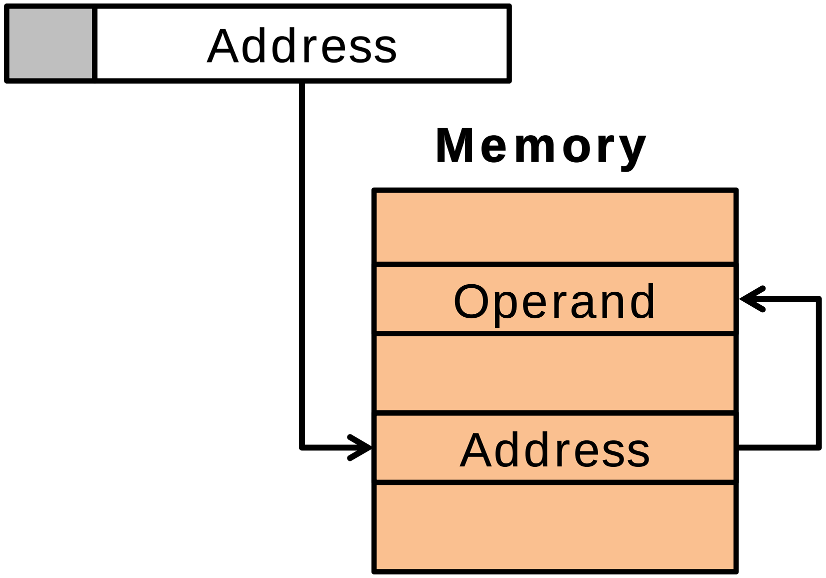
Addressing modes: a very GENERAL introduction.3

Indirect addressing

Indirect addressing

instruction contains a (short) address in memory which in turn contains the address of operand

load R1, [0xC000]

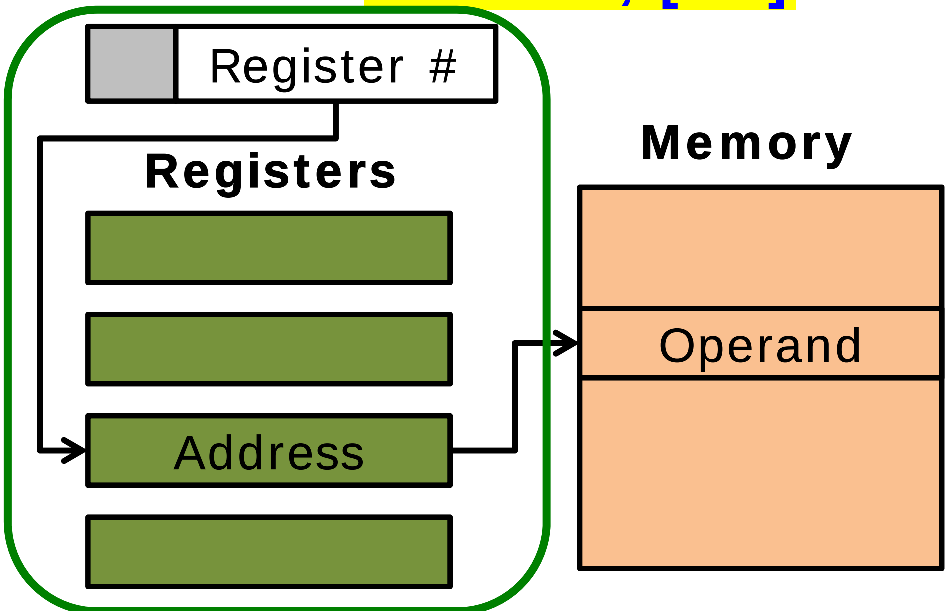


Register Indirect addressing (Index, Based)

- instruction contains a register number
- register contains address of the operand

CPU

store R1, [R2]



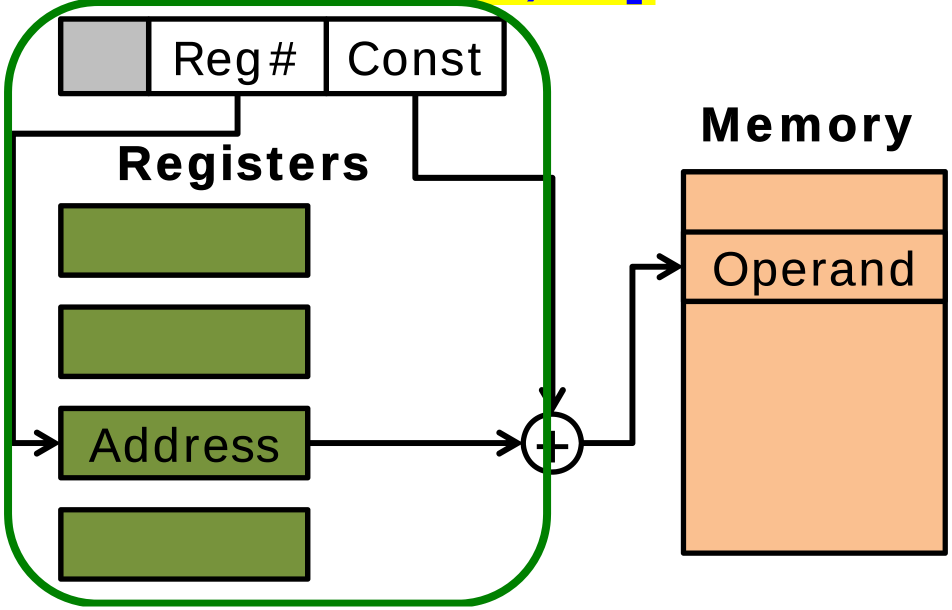
Addressing modes: a very GENERAL introduction.4

Register Indirect addressing with offset/displacement

Based/Indexed with immediate offset

- instruction contains a register number and a constant
- register contains address of the operand and offset is added to it to get EA

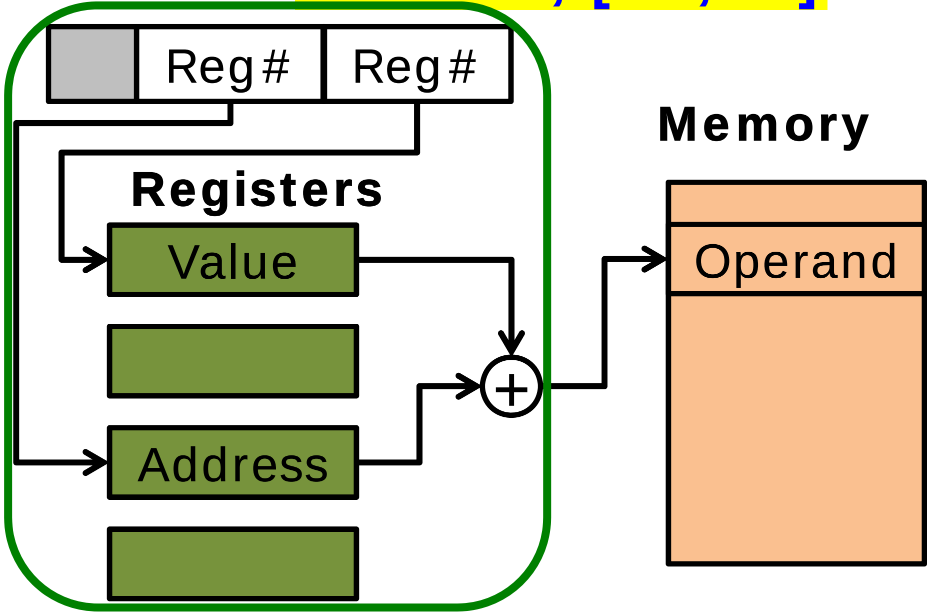
CPU **load R1, [#4,R1]**



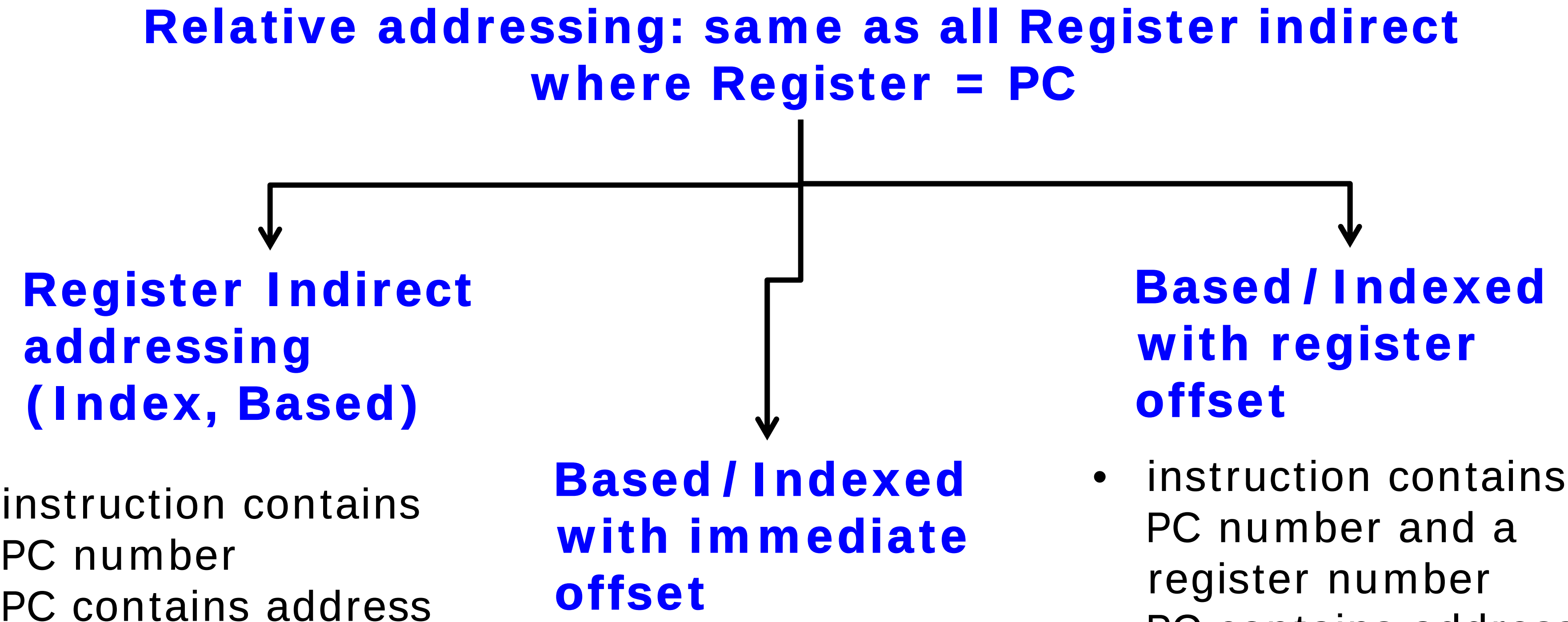
Based/Indexed with register offset

- instruction contains two register numbers
- one register contains address of the operand and the other the offset

CPU **store R1, [R8,R2]**



Addressing modes: a very GENERAL introduction.5



Relative addressing: same as all Register indirect where Register = PC

Register Indirect addressing (Index, Based)

- instruction contains PC number
- PC contains address of the operand

Based / Indexed with immediate offset

- instruction contains PC number and a constant
- PC contains address of the operand and offset is added to it to get EA

Based / Indexed with register offset

- instruction contains PC number and a register number
- PC contains address of the operand and the other register contains the offset

Addressing modes: a very GENERAL introduction

Immediate addressing

instruction contains the value of the operand

```
move R1, #3
```

Absolute addressing

```
load R1, 0xC000
```

instruction contains the full binary address of operand

Register {direct} addressing

```
add R1, R2
```

operand =content of the register, instruction contains the register number

Based or indexed or indirect addressing

address of operand = content of register,
instruction contains the register number

```
load R1,[R2]
```

Based or indexed addressing with immediate offset

address of operand = register + immediate offset, both contained in instruction

```
load R1,#4[R2]
```

Based or indexed addressing with register offset

address of operand = register + offset in register

```
load R1,[R2,R3]
```

Relative addressing

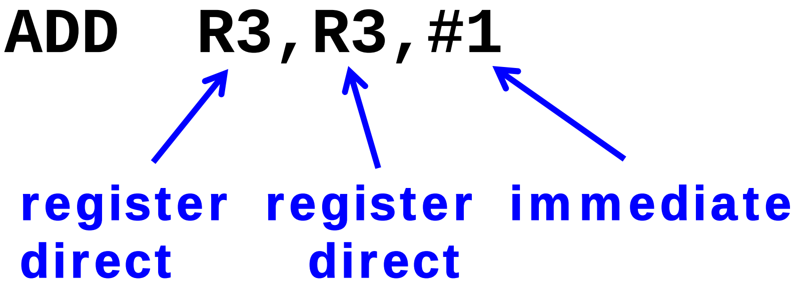
```
load R1,#8[PC]
```

address of operand = as above with PC as base register

Examples of addressing modes (ARM): *immediate* and *register direct*

Instruction

Semantics (what does it do?)



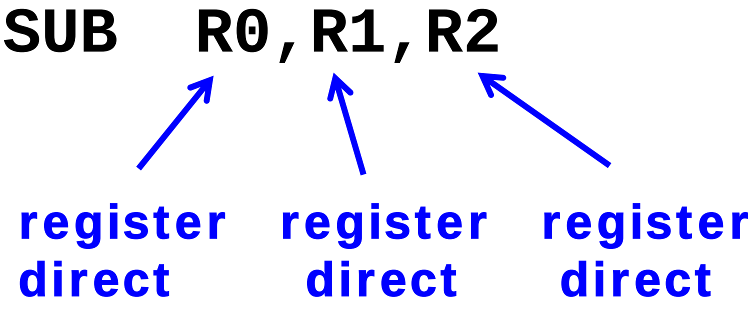
```
ADD   R3,R3,#1
```

@R3 = R3+1

register direct

register direct

immediate



```
SUB   R0,R1,R2
```

@R0 = R1-R2

register direct

register direct

register direct

```
ADD   R0,[R1,#4]
```

not allowed – only with registers, not with memory

Examples of addressing modes (ARM):
based/indexed with or without offset

Instruction	Semantics (what does it do?)
`MOV R0,[R1]`	*not allowed – MOV only with registers, not with memory*
`LDR R0,[R2]`	@ R0 ← mem[R2]
`MOV R3,R2`	@ R3 ← R2
`STR R0,[R1,#4]`	@ R0 → mem[R1+4]

`LDR R0,[R2]`: R0 – register direct; [R2] – register based/indexed/indirect

`MOV R3,R2`: register , register

`STR R0,[R1,#4]`: R0 – register; [R1,#4] – register based/indexed with immediate offset pre-indexed

Examples of addressing modes (ARM):
based/indexed with/without offset

Instruction — **Semantics (what does it do?)**

LDR R0,[R1,#4] — @ R0 ← mem[R1+4]

register, register based/indexed with immediate offset pre-index

ADD R0,[R1,#4] — *not allowed – only with registers, not with memory*

MOV R0,#0x23 — @ R0 ← 35_{10}

register, immediate

LDR R0,[R1,R2] — @ R0 ← mem[R1+R2]

register, register indexed with register offset pre-index

Examples of addressing modes (ARM): *based/indexed with/without offset*

Instruction

Semantics
(what does it do?)

```
LDR  R0,[R1,#4]!        @ R1 = R1+4
                        and then R0 ← mem[R1]
```

register, register based/indexed with
immediate offset pre-index *and writeback*

```
LDR  R0,[R1,#4]         @ R0 ← mem[R1+4]
```

register, register based/indexed with
immediate offset pre-index

```
LDR  R0,[R2],#8         @ R0 ←mem[R2] and
                        then R2 = R2 + 8
```

register, register indexed with
immediate offset post-index

```
LDR  R0,[R2],R4         @ R0 ←mem[R2] and
                        then R2 = R2 + R4
```

register, register indexed with
register offset post-index

Summary

Addressing modes = set of semantics and syntactic modes
which can be used for operands

- ❑ The various addressing modes defined in a given ISA define how machine language instructions identify the operands.
  - ➢ This is how an instructions is “decoded” for execution in the Fetch-Decode-Execute cycle.
- ❑ An addressing mode specifies how to calculate the *Effective Memory Address* (EA) of an operand by using information held within the machine instruction itself.

(paraphrased from Wikipedia)

at times the textbook uses it as “ways of computing the address of a value in memory” in a looser way

Addressing modes in ARM: the differences

Immediate addressing

instruction contains the value of the operand ➔ ADD R1,R1,#3

Absolute addressing (aka direct addressing)

instruction contains the address of operand ➔ NO!

Indirect addressing

Instruction contains address of location in memory which in turn contains the address of operand ➔ NO!

Register or Register Direct

register contains operand ➔ ADD R1,R2,R3

Register Indexed (or register indirect/based) with no offset

register contains address of operand ➔ LDR R1,[R2] only load/store

Register Indexed (etc.) with immediate offset pre-indexed or post-indexed

address of operand = register + offset ➔ LDR R1,[R2],#4

or STR R1,[R3,#4] only load/store

Register Indexed (etc.) with register offset pre-indexed or post-indexed

address of operand = register+register ➔ LDR R1,[R2,R3]

or STR R1,[R2],R3 only load/store

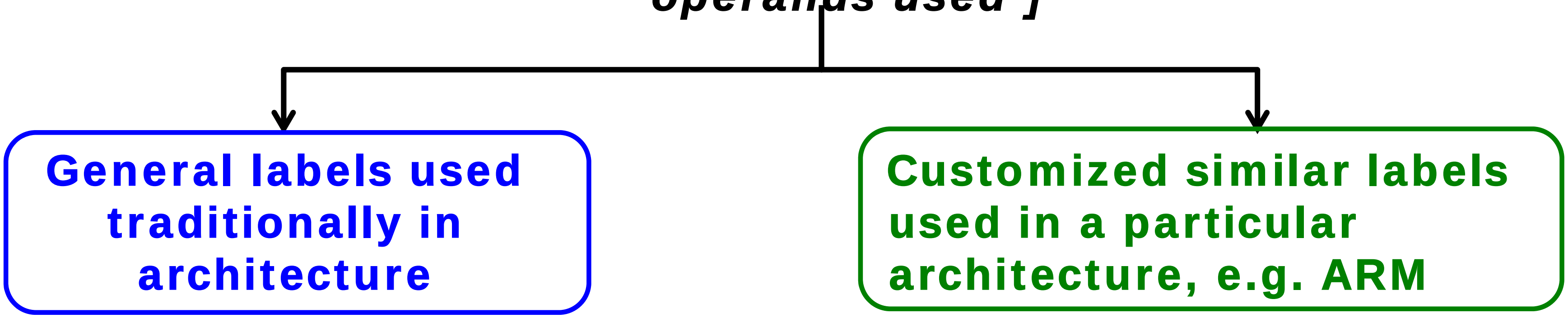
Relative addressing *(same as Register Indexed, but register is PC)*

address of operand = PC + offset ➔ YES

Clear up some confusion

Addressing modes = set of semantics and syntactic modes which can be used for operands

[a way to encode inside each instruction the type of operands used]



General labels used traditionally in architecture

Customized similar labels used in a particular architecture, e.g. ARM

e.g. immediate,
direct,
absolute,
indirect,
register,
register indirect,
register based,
register indexed,
with offset,
with displacement

e.g. immediate,
register, register direct
register indexed
with variations

ARM offers a broad range of addressing modes, especially for a RISC processor. ARM's addressing modes are hard to summarize in a few sentences. Examples follow.

ARM Addressing Modes once more .1

#constant	Immediate (small constant inside the Instruction itself)	ADD R1,R2,#4 MOV R3,#0x1C
Rn	Register (no memory involvement)	ADD R1,R2,#4 SUB R3,R4,R2 MOV R4,R1
[Rn]	Register Indexed {Based} (Load/Store only)	LDR R1,[R2] STR R2,[R3]

ARM Addressing Modes once more.2

Register Indexed {Based} with Immediate Offset (Load/Store only)

[Rn,#offset]	Pre-Indexed	LDR R1,[R2,#4] STR R2,[R3,#8]
[Rn,#offset]!	Pre-Indexed with writeback	LDR R1,[R2,#4]! STR R2,[R3,#8]!
[Rn],#offset	Post-Indexed	LDR R1,[R2],#-4 STR R2,[R3],#8

Examples:

LDR R1,[R2,#4]	➔ Load into R1 the content of memory at address = R2+4
STR R2,[R3,#8]!	➔ Store the value in R2 into memory at address = R3+8 and after assign R3=R3+8
LDR R1,[R2],#-4	➔ Load into R1 the content of memory at address = R2 and after assign R2=R2-4

ARM Addressing Modes once more.3

Register Indexed with register offset (Load/Store only)

[Rn,Rm]	Pre-Indexed	LDR R1,[R2,R3] STR R2,[R3,R1]
[Rn,Rm]!	Pre-Indexed with writeback	LDR R1,[R2,R3]! STR R2,[R3,R1]!
[Rn],Rm	Post-Indexed	LDR R1,[R2],R3 STR R2,[R3],R1

Examples:

LDR R1,[R2,R3] ➔ Load into R1 the content of memory at address = R2+R3

STR R2,[R3,R1]! ➔ Store the value in R2 into memory at address = R3+R1 and after assign R3=R3+R1

LDR R1,[R2],R3 ➔ Load into R1 the content of memory at address = R2 and after assign R2=R2+R3

ARM Addressing Modes once more.4

Register Indexed with scaled register offset (Load/Store only)

[Rn,Rm,*shift*]	Pre-Indexed	LDR R1,[R2,R3] STR R2,[R3,R1,ASL #2]
[Rn,Rm,*shift*]!	Pre-Indexed with writeback	LDR R1,[R2,R3]! STR R2,[R3,R1,ASL #2]!
[Rn],Rm,*shift*	Post-Indexed	LDR R1,[R2],R3 STR R2,[R3],R1,ASR #2

Examples:

STR R2,[R3,R1,ASL#2]! ➔ Store the value in R2 into memory at address = R3+R1*4 and after assign R3=R3+R1*4

Thoughts

1. How many bits in each instruction should be reserved for opcode?

2. Which operands should be in registers and which in memory?

3. How many operands is a “better” design?

4. What is one trying to optimize?

5. ARM is a RISC architecture. What does it imply for instruction formats and addressing modes?

RISC ⇒ only load and store instructions access memory; other instructions like add & subtract have only register or immediate operands.

Look at some instructions and their execution

C = A + B; /*in a high level language*/

Get address of A into a register, e.g. R1

Get *content* of A into a register, e.g. R2

Get address of B into a register, e.g. R3

Get *content* of B into a register, e.g. R4

Add R2 and R4 and result in R5

Get address of C into a register, e.g. R6

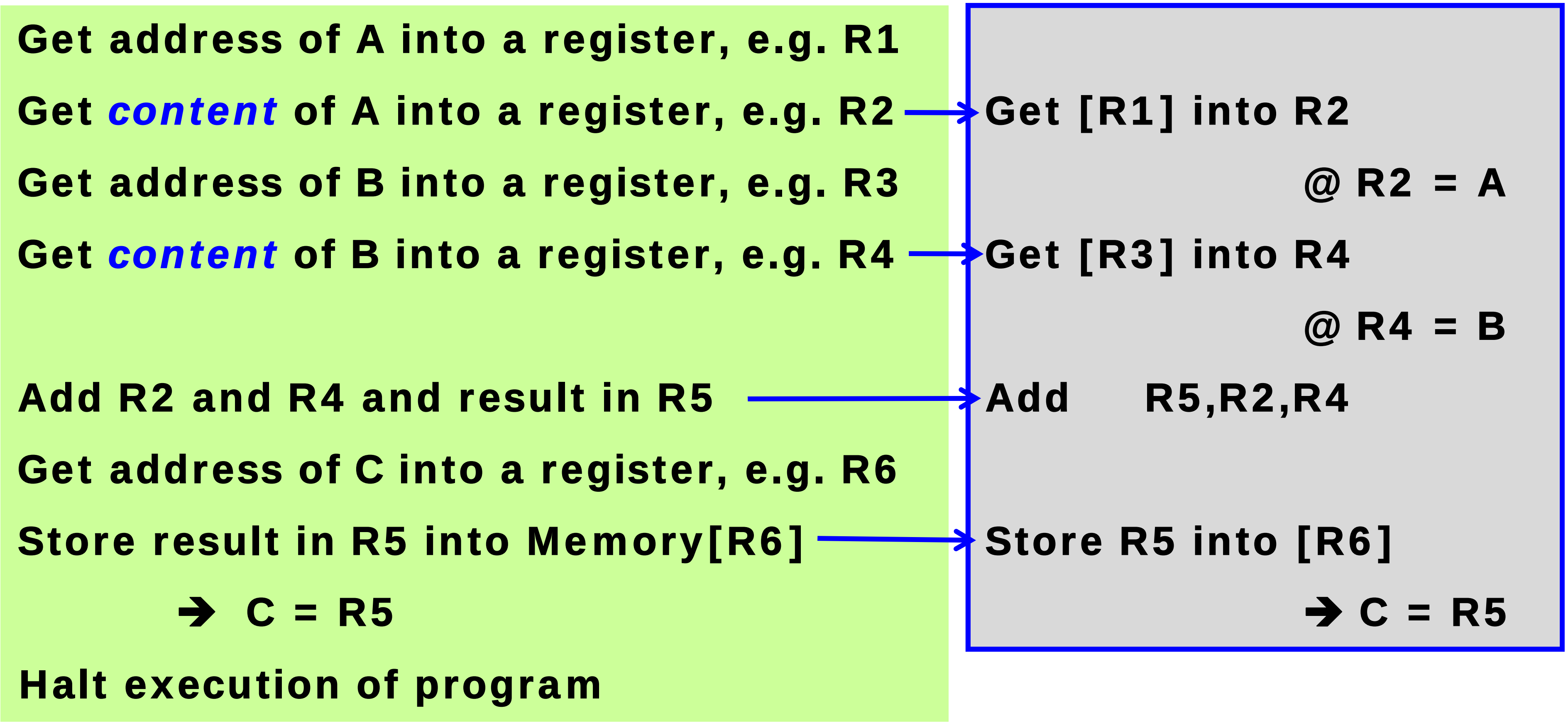
Store result in R5 into Memory[R6]

➔ C = R5

Halt execution of program

Look at some instructions and their execution

C = A + B; /*in a high level language*/



Get address of A into a register, e.g. R1

Get *content* of A into a register, e.g. R2

Get address of B into a register, e.g. R3

Get *content* of B into a register, e.g. R4

Add R2 and R4 and result in R5

Get address of C into a register, e.g. R6

Store result in R5 into Memory[R6]

➔ C = R5

Halt execution of program

Get [R1] into R2

@ R2 = A

Get [R3] into R4

@ R4 = B

Add R5,R2,R4

Store R5 into [R6]

➔ C = R5

Look at some instructions and their execution

C = A + B; /*in a high level language*/

	In ARM
Get address of A into R1	LDR R1,=A
Get [R1] into R2 @ R2 = content of A	LDR R2,[R1]
Get address of B into R3	LDR R3,=B
Get [R3] into R4 @ R4 = content of B	LDR R4,[R3]
Add R5,R2,R4	ADD R5,R2,R4
Get address of C into R6	LDR R6,=C
Store R5 into [R6] @ C = R5	STR R5,[R6]
Halt execution of program	SWI 0x11

Assume the program is loaded into memory at address 0x0000C000

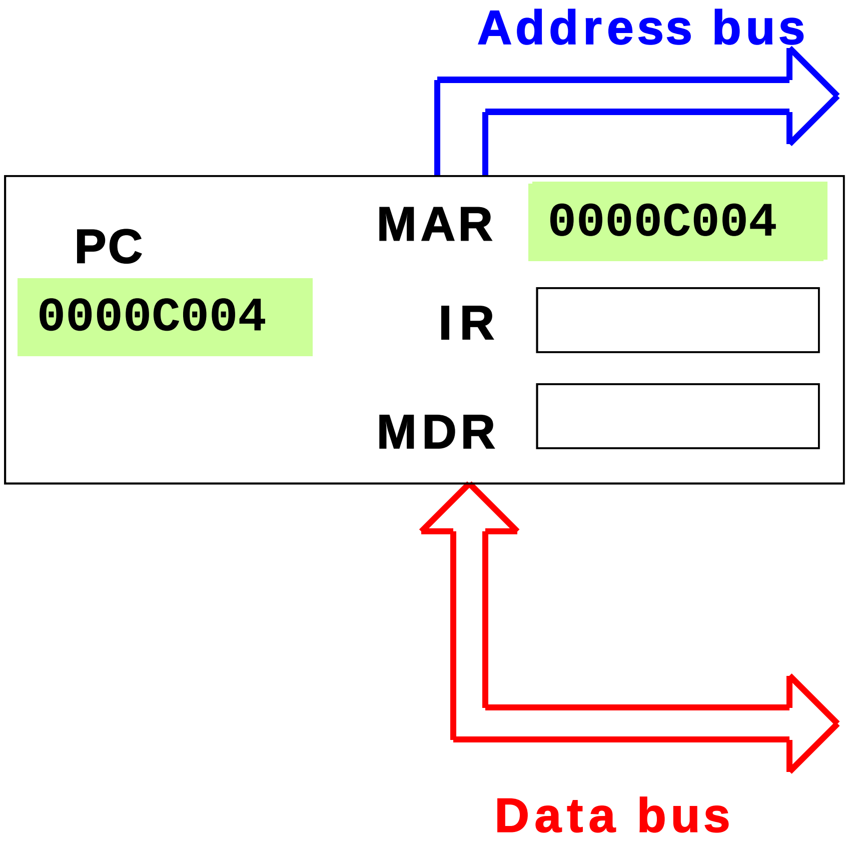
```
LDR     R1,=A
LDR     R2,[R1]
LDR     R3,=B
LDR     R4,[R3]
ADD     R5,R2,R4
LDR     R6,=C
STR     R5,[R6]
SWI     0x11
```

```
LDR     R1,=A              0000C000
LDR     R2,[R1]            0000C004
LDR     R3,=B              0000C008
LDR     R4,[R3]            0000C00C
ADD     R5,R2,R4           0000C010
LDR     R6,=C              0000C014
STR     R5,[R6]            0000C018
SWI     0x11               0000C01C

A:      .word  5           0000C02C
B:      .word  6           0000C030
C:      .word  0           0000C034
```

Snapshot of memory after loading code

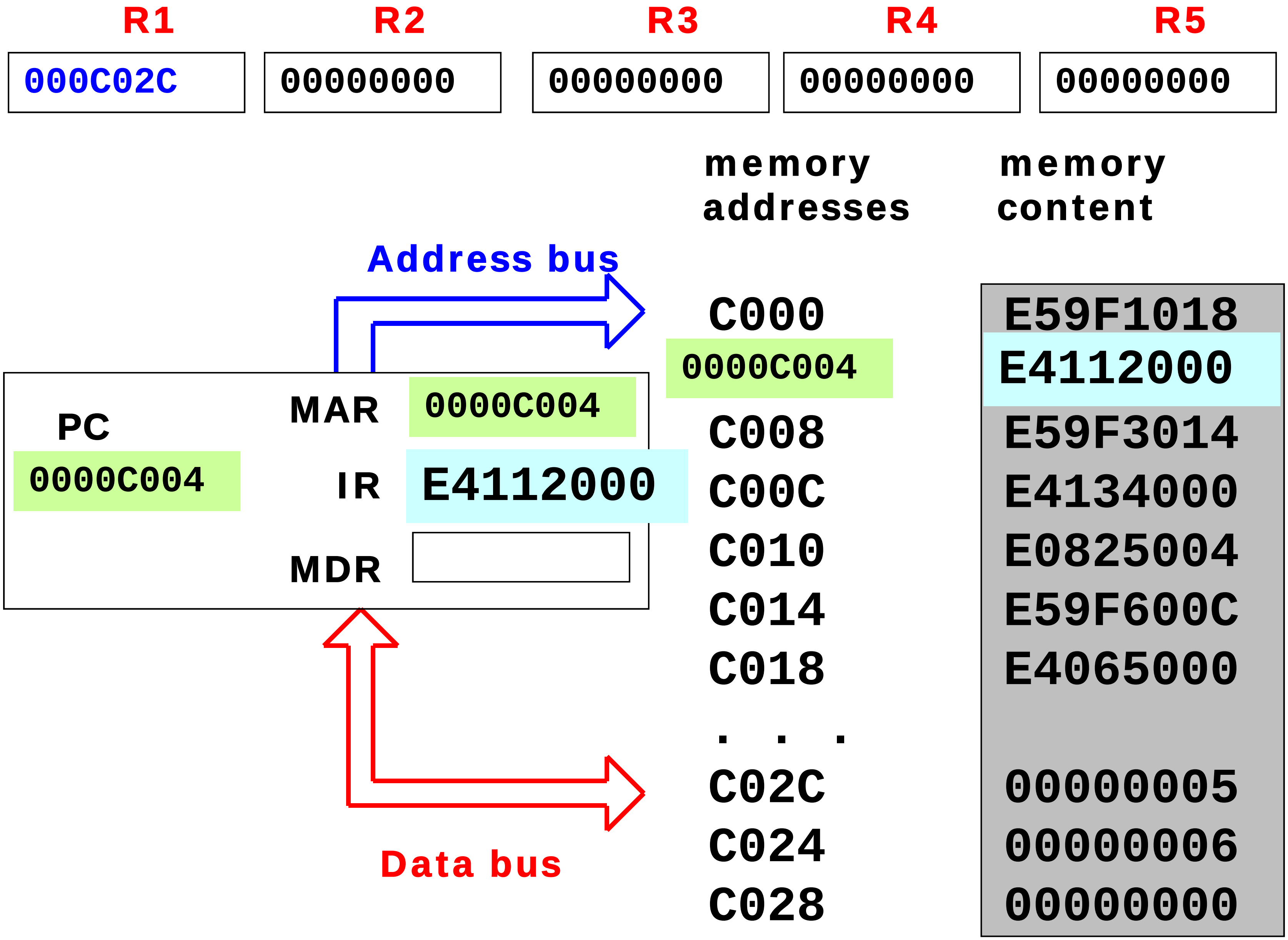
```
                               memory      memory
                               addresses   content

LDR  R1,=A                     0000C000    E59F1018
LDR  R2,[R1]                   0000C004    E4112000
LDR  R3,=B                     0000C008    E59F3014
LDR  R4,[R3]                   0000C00C    E4134000
ADD  R5,R2,R4                  0000C010    E0825004
LDR  R6,=C                     0000C014    E59F600C
STR  R5,[R6]                   0000C018    E4065000
SWI  0x11                      0000C01C    EF000011
. . .
A  .word   5                   0000C02C    00000005
B  .word   6                   0000C030    00000006
C  .word   0                   0000C034    00000000
```

R1	R2	R3	R4	R5
000C02C	00000000	00000000	00000000	00000000



memory addresses	memory content
0000C000	E59F1018
0000C004	E4112000
0000C008	E59F3014
0000C00C	E4134000
0000C010	E0825004
0000C014	E59F600C
0000C018	E4065000
. . .	
0000C02C	00000005
0000C024	00000006
0000C028	00000000

```
LDR   R2,[R1]
```

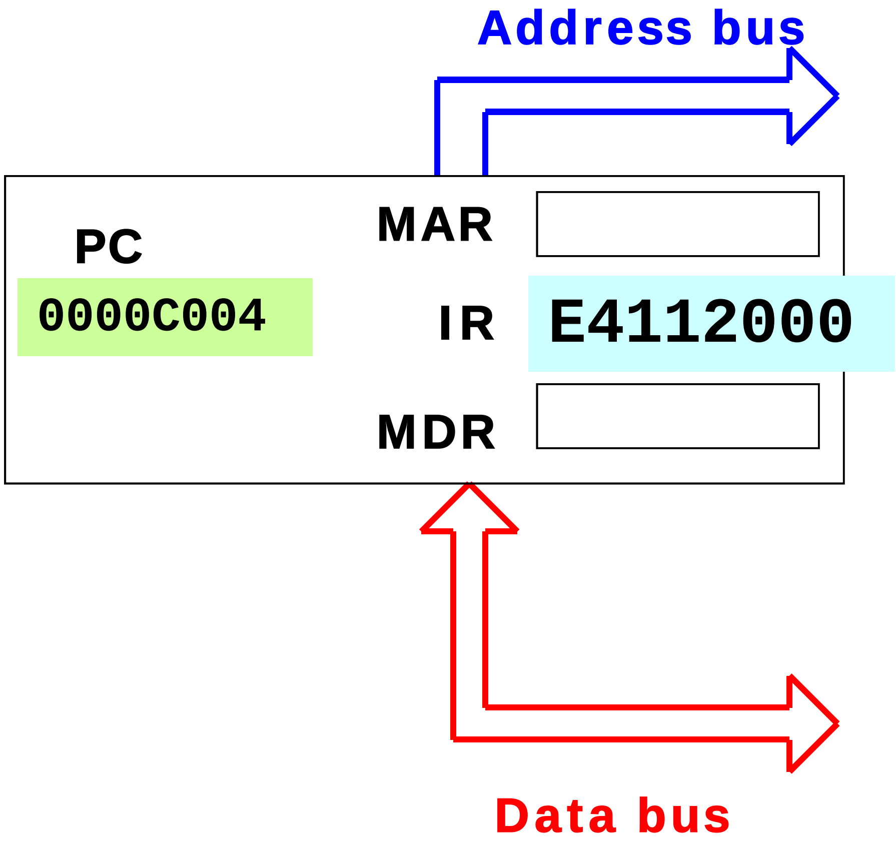


R1	R2	R3	R4	R5
000C02C	00000000	00000000	00000000	00000000

memory addresses	memory content
C000	E59F1018
0000C004	E4112000
C008	E59F3014
C00C	E4134000
C010	E0825004
C014	E59F600C
C018	E4065000
. . .	
C02C	00000005
C024	00000006
C028	00000000

```
LDR   R2,[R1]
```

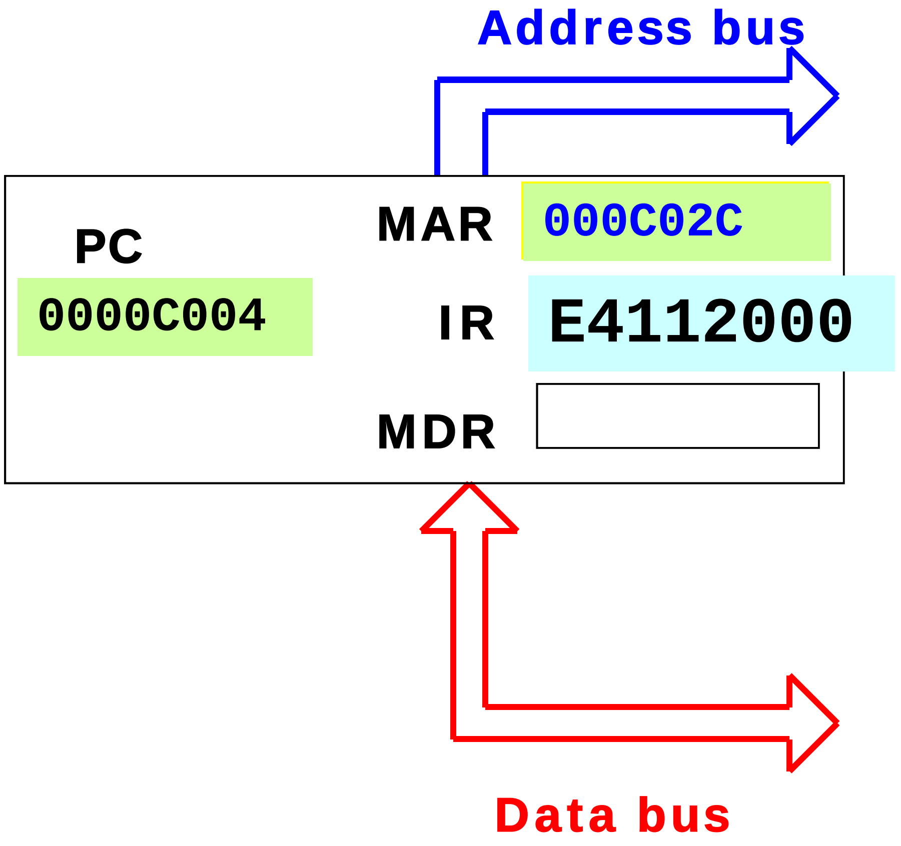
R1	R2	R3	R4	R5
000C02C	00000000	00000000	00000000	00000000



memory addresses	memory content
C000	E59F1018
C004	E4112000
C008	E59F3014
C00C	E4134000
C010	E0825004
C014	E59F600C
C018	E4065000
. . .	
C02C	00000005
C024	00000006
C028	00000000

```
LDR  R2,[R1]
```

R1	R2	R3	R4	R5
000C02C	00000005	00000000	00000000	00000000



Address bus

PC 0000C004

MAR 000C02C

IR E4112000

MDR

Data bus

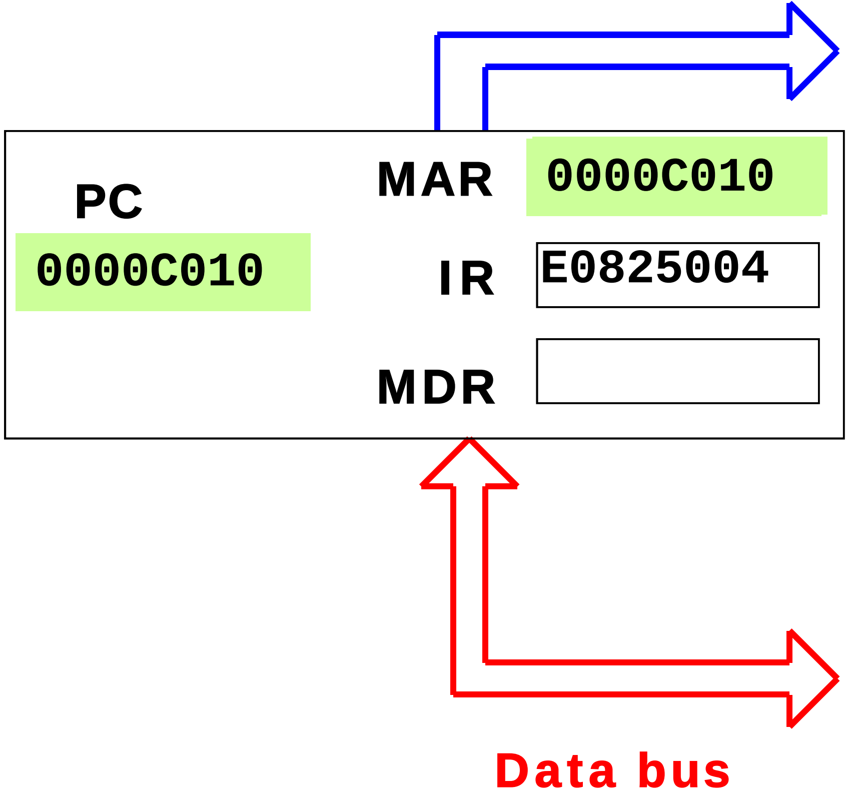
memory addresses	memory content
C000	E59F1018
C004	E4112000
C008	E59F3014
C00C	E4134000
C010	E0825004
C014	E59F600C
C018	E4065000
. . .	
C02C	00000005
C024	00000006
C028	00000000

```
LDR  R2,[R1]
```

Snapshot of memory after loading code

	memory addresses	memory content
LDR R1,=A	C000	E59F1018
LDR R2,[R1]	C004	E4112000
LDR R3,=B	C008	E59F3014
LDR R4,[R3]	C00C	E4134000
ADD R5,R2,R4	C010	E0825004
LDR R6,=C	C014	E59F600C
STR R5,[R6]	C018	E4065000
. . .		
A .word 5	C02C	00000005
B .word 6	C030	00000006
C .word 0	C034	00000000

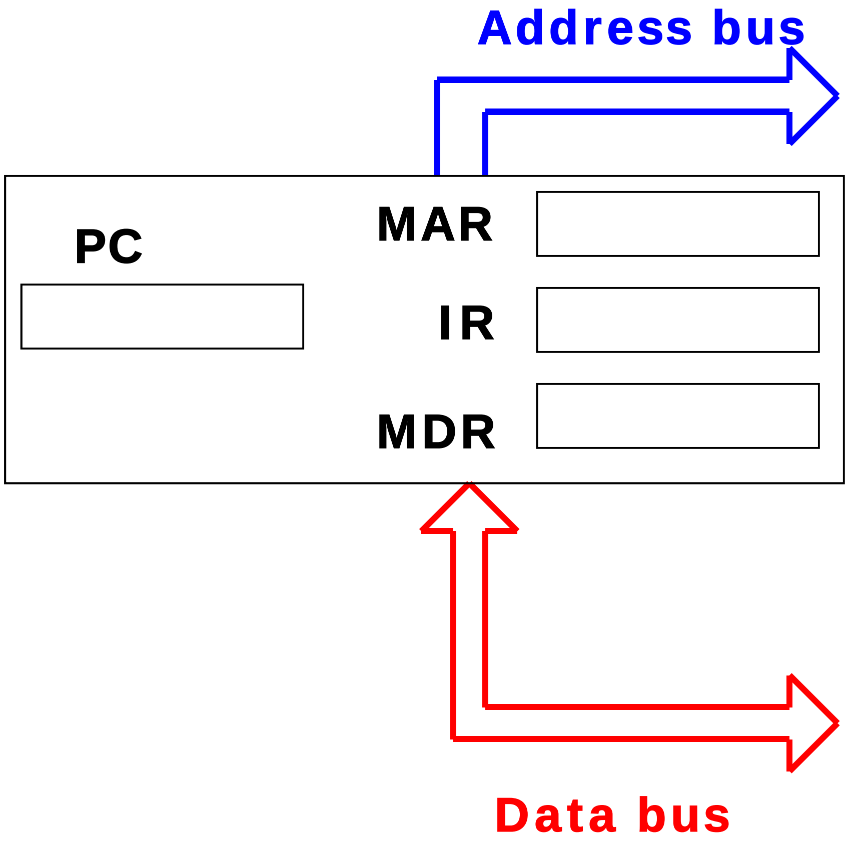
R1	R2	R3	R4	R5
000C02C	00000005	0000C020	00000006	0000000B 00000000



memory addresses	memory content
C000	E59F1018
C004	E4112000
C008	E59F3014
C00C	E4134000
C010	E0825004
C014	E59F600C
C018	E4065000
C01C	00000005
C020	00000006
C024	00000000

```
ADD   R5,R2,R4
```

R1	R2	R3	R4	R5
000C02C	00000005	0000C020	00000006	00000000



Address bus

PC

MAR

IR

MDR

Data bus

memory addresses	memory content
C000	E59F1018
C004	E4112000
C008	E59F3014
C00C	E4134000
C010	E0825004
C014	E59F600C
C018	E4065000
C01C	00000005
C020	00000006
C024	00000000

```
ADD   R5,R2,R4
```

Relative addressing mode: one useful common example only

In order to copy into a register the content of a location in memory, we need two instructions:

```
LDR   R1,=temp
LDR   R1,[R1]    @ here R1 = temp = 3
.  .  .  .  .
     .DATA
temp .word 3     @ declaration and
                 @ initialization of temp
```

Addressing mode is PC RELATIVE addressing

Why?

Small Summary for ARM Syntax so far (1)

(a) Assigning constants

```
MOV     Ri,#number   @ if number < 255 (and some other cases)
             MOV     r1,#15          @ r1 = 15
             MOV     r1,#0x000F
```

(b) Assigning BIG constants

```
LDR     Ri,=number   @ works for any 32 bit number
             LDR     r1,=15000       @ r1 = 15000
```

(c) Loading addresses of variables

```
LDR     Ri,=VariableName
             LDR     r1,=var1        @ r1 = address of var1
```

(d) Loading the content of a word variable (2 instructions) from memory

```
LDR     r1,=NUM      @ r1 = address of NUM
LDR     r1,[r1]      @ r1 = content of NUM
```

In general:

```
LDR     Ri,[Rj] @ where Rj already has the address
```

Small Summary for ARM Syntax so far (2)

(e) Copy between registers

MOV Ri, Rj

```
MOV    r1,r2          @ r1 = r2
```

(f) Storing the content of a variable (2 instructions) into memory

LDR Ri, =VariableName @ Ri has the address

STR Rj, [Ri] @ write at the address in Ri

```
LDR    r1,=NUM        @ r1 = address of NUM
STR    r2,[r1]        @ NUM = content of r2
```

(g) Loading the content of a character (2 instructions)

```
LDR    r1,=MyChar    @ r1 = address of MyChar
LDRB   r1,[r1]       @ r1 = byte content of MyChar
```

In general:

LDRB Ri,[Rj] @ where Rj already has the address

Addressing modes: a very GENERAL introduction

Immediate addressing

DEPRECATED

move R1, #3

instruction contains the value of the operand

load R1, 0xC000

Absolute addressing

instruction contains the full binary address of operand

Indirect addressing

load R1, [0xA020]

instruction contains the binary address of a location
in memory which in turn contains the address of operand

Add R1, R2

Register {direct} addressing

register contains operand, instruction contains the register number

Based or indexed addressing with offset

address of operand = register + offset, both contained in instruction

load R1,#4[R2]

Relative addressing

load R1,#8[PC]

address of operand = PC + offset