CSCU9V4
Systems

Systems lecture 9

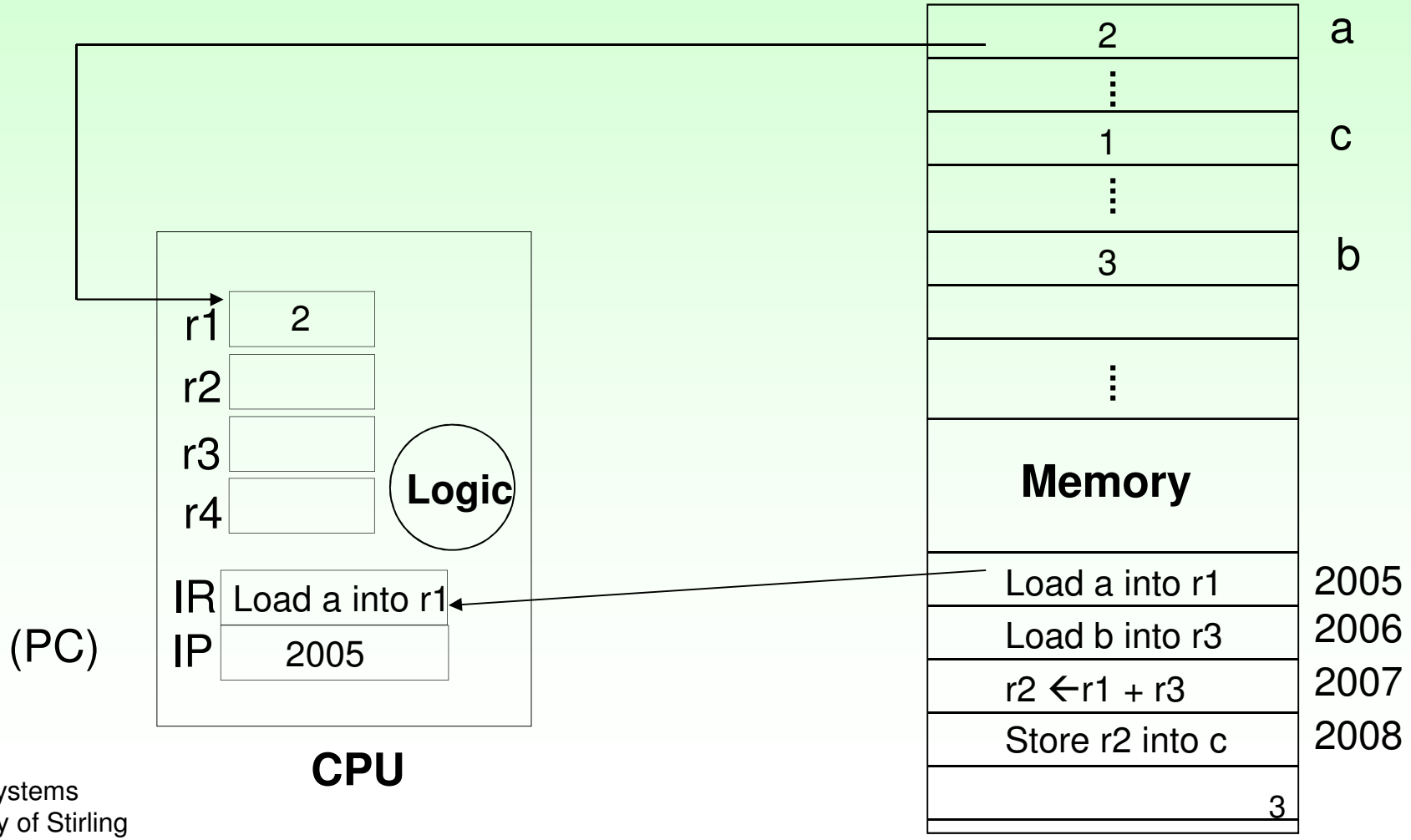
Computer Organisation 4

Instructions

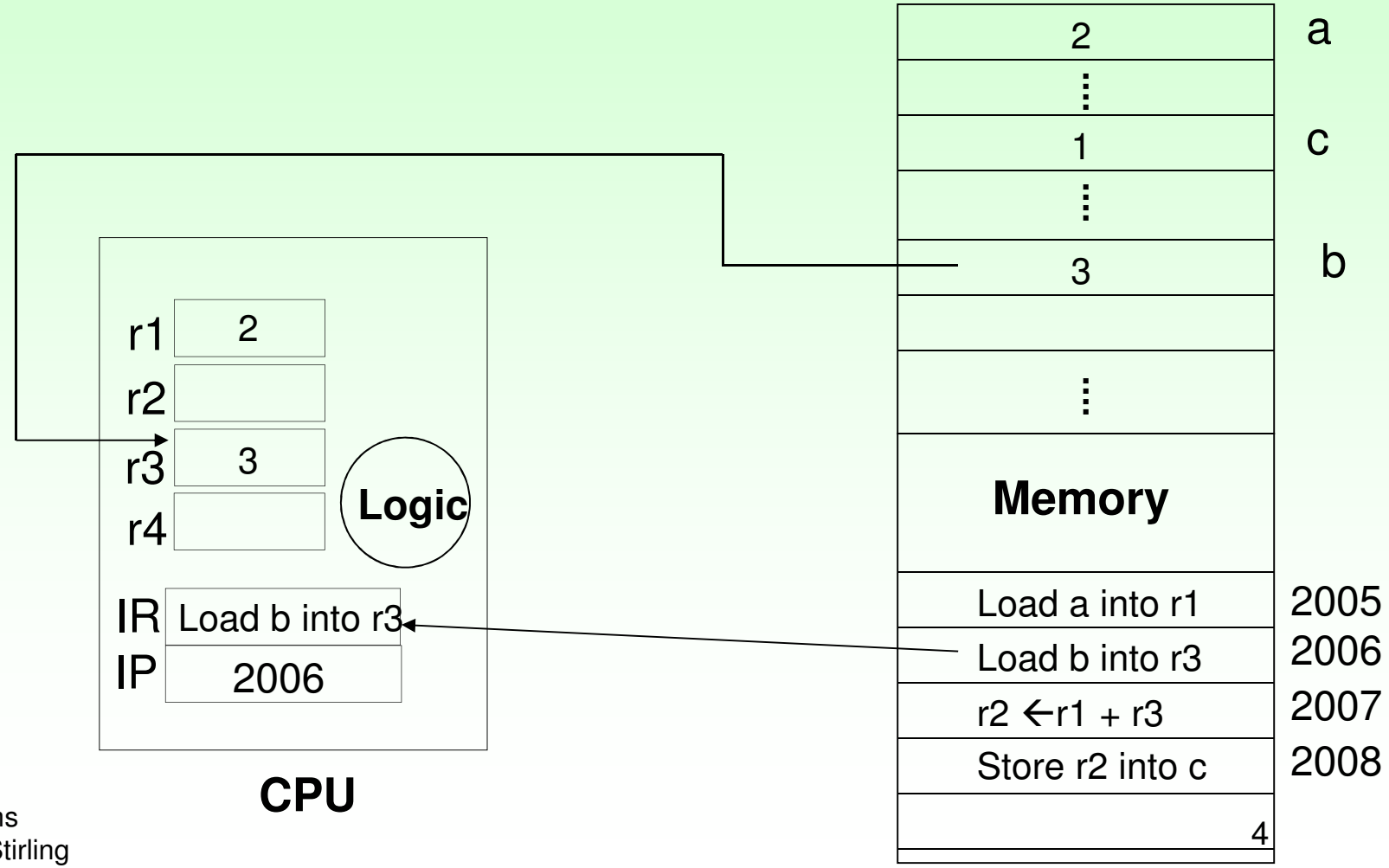
A Simple Program

- Want to add values of variables a and b (assumed to be in memory), and put the result in variable c in memory, I.e. c ← a+b
- Instructions in program
  - Load a into register r1
  - Load b into register r3
  - r2 ← r1 + r3
  - Store r2 in c
- More precisely,
  - LOAD R1, A (some representation of A, i.e. hex address or equiv)
  - LOAD R3, B
  - ADD R2, A, B
  - STORE R2, C

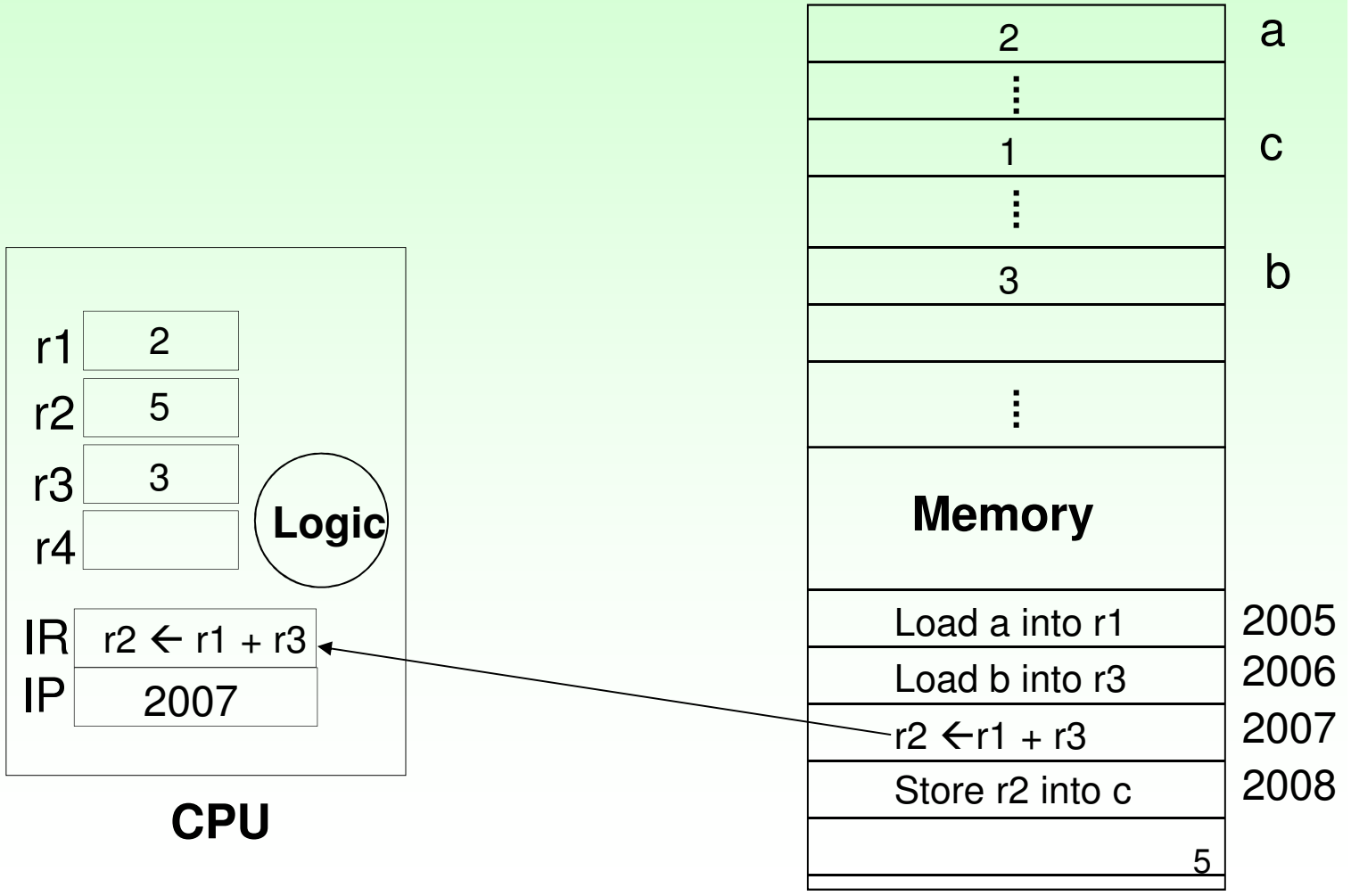
Running the Program



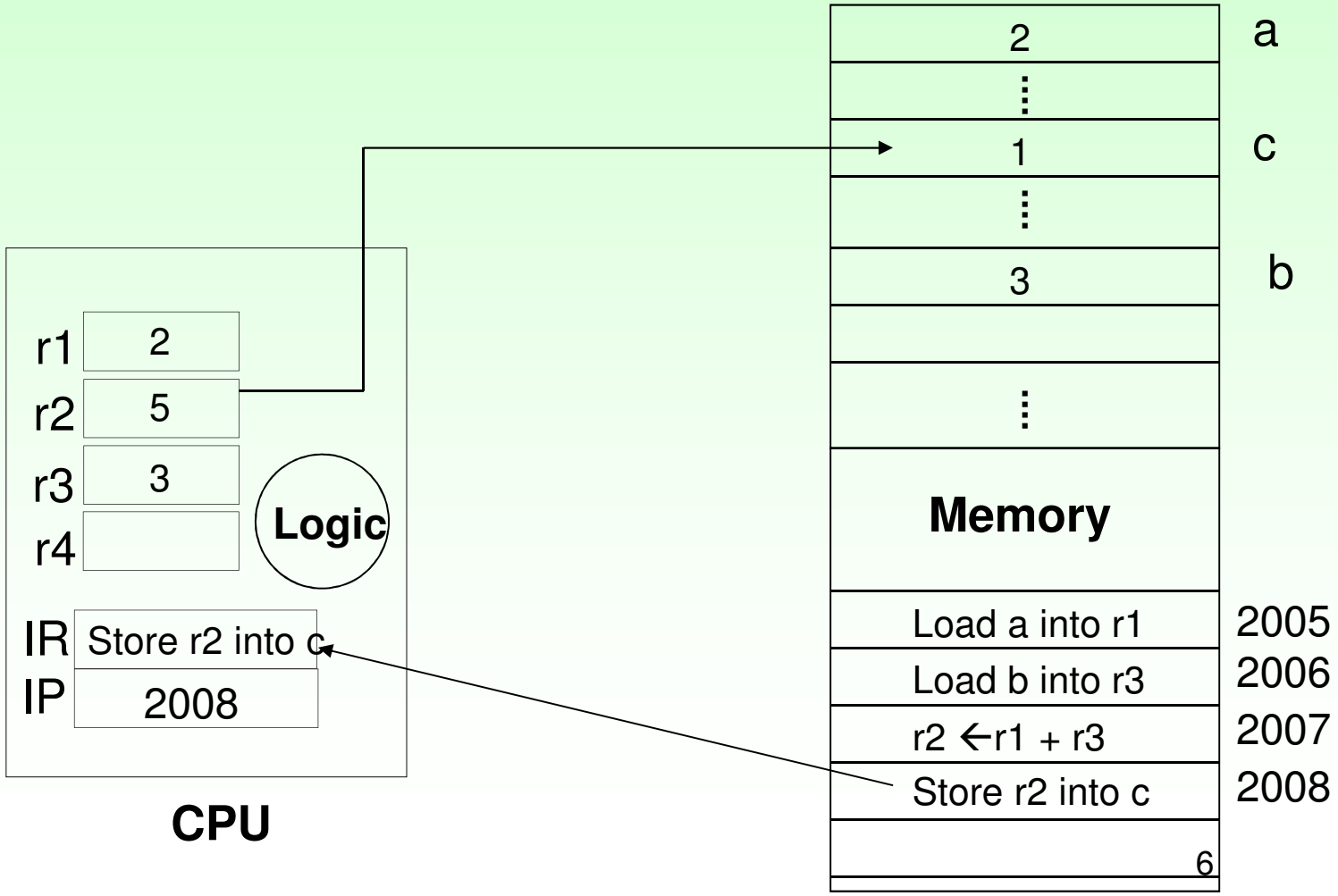
Running the Program



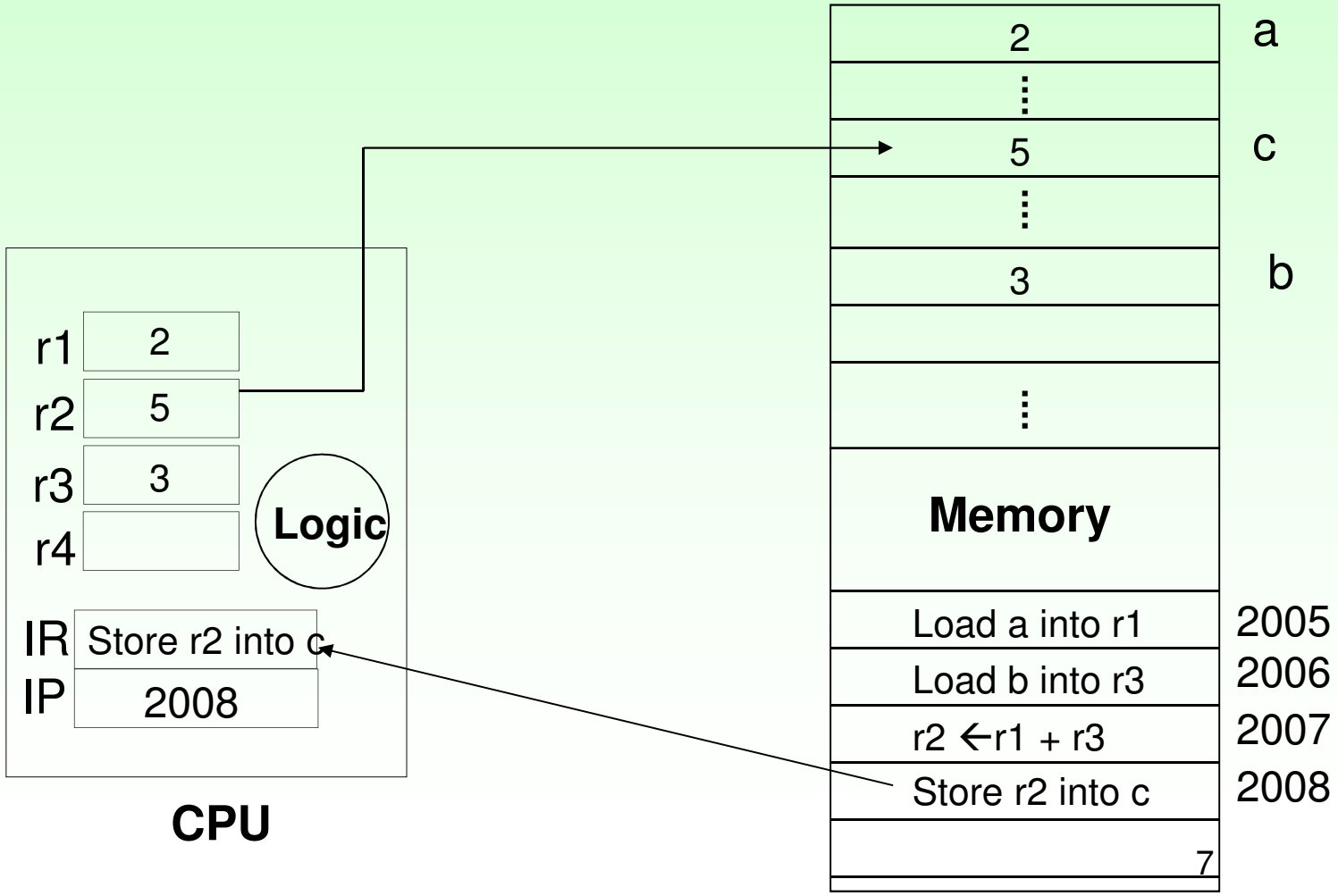
Running the Program



Running the Program

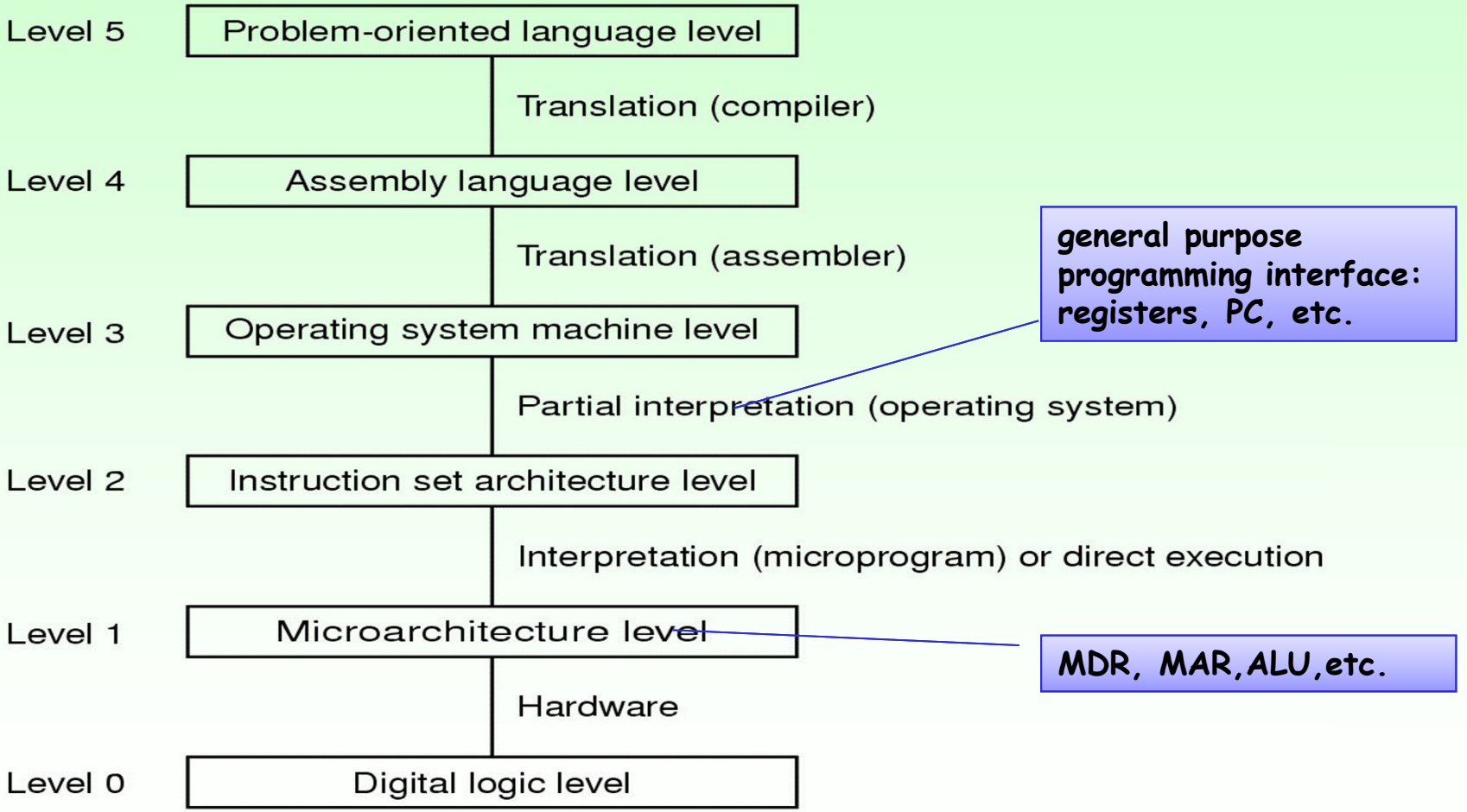


Running the Program



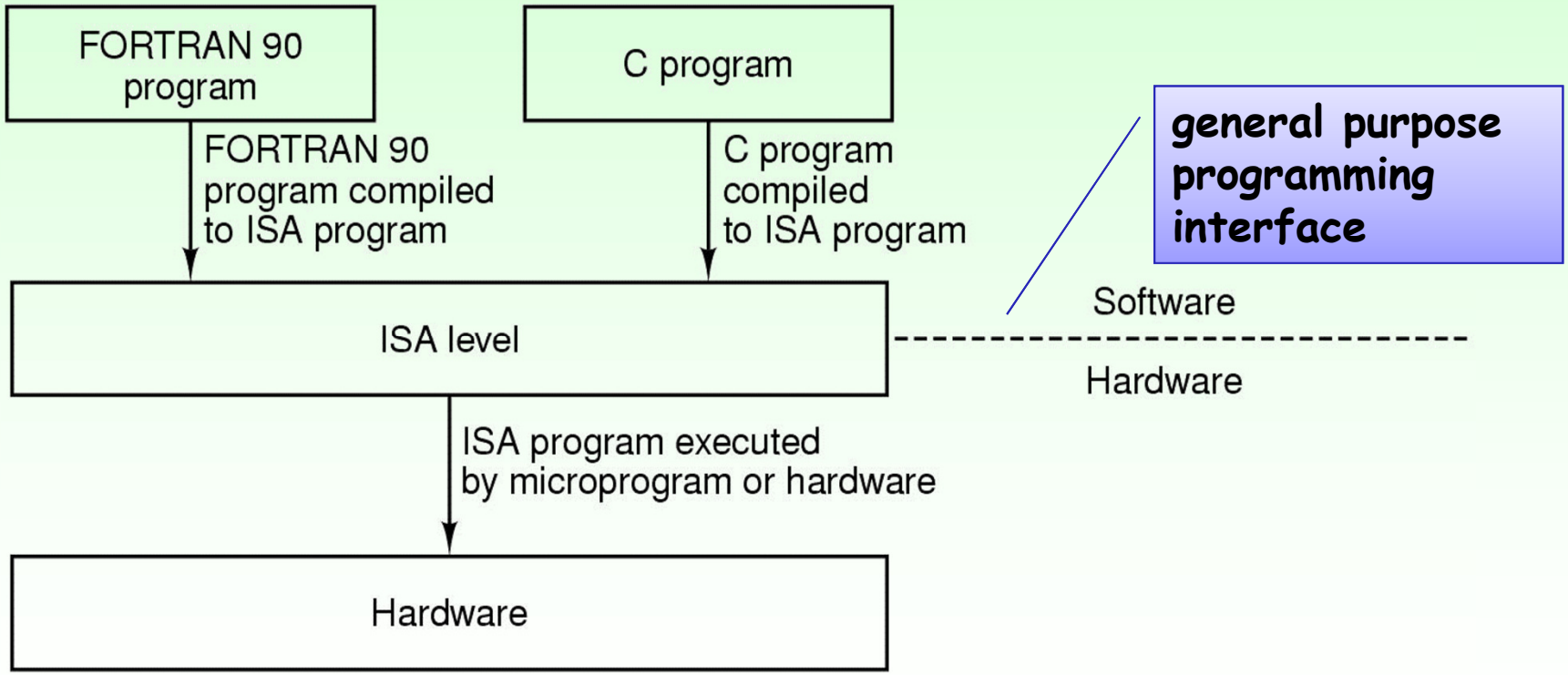
The Instruction Level

The instruction level is the lowest level programming interface to the CPU.



The Instruction Level

- The hardware / software boundary
- ISA may last 20 years (or more!)
- Should have a clean set of instructions for the compiler designers
- Hardware designers will want to change the underlying hardware



The Instruction Level

ISA should hide the lower levels

- in practice this is often not true
- to exploit new processor features may require changes to the ISA

If the ISA changes they aim for backwards compatibility

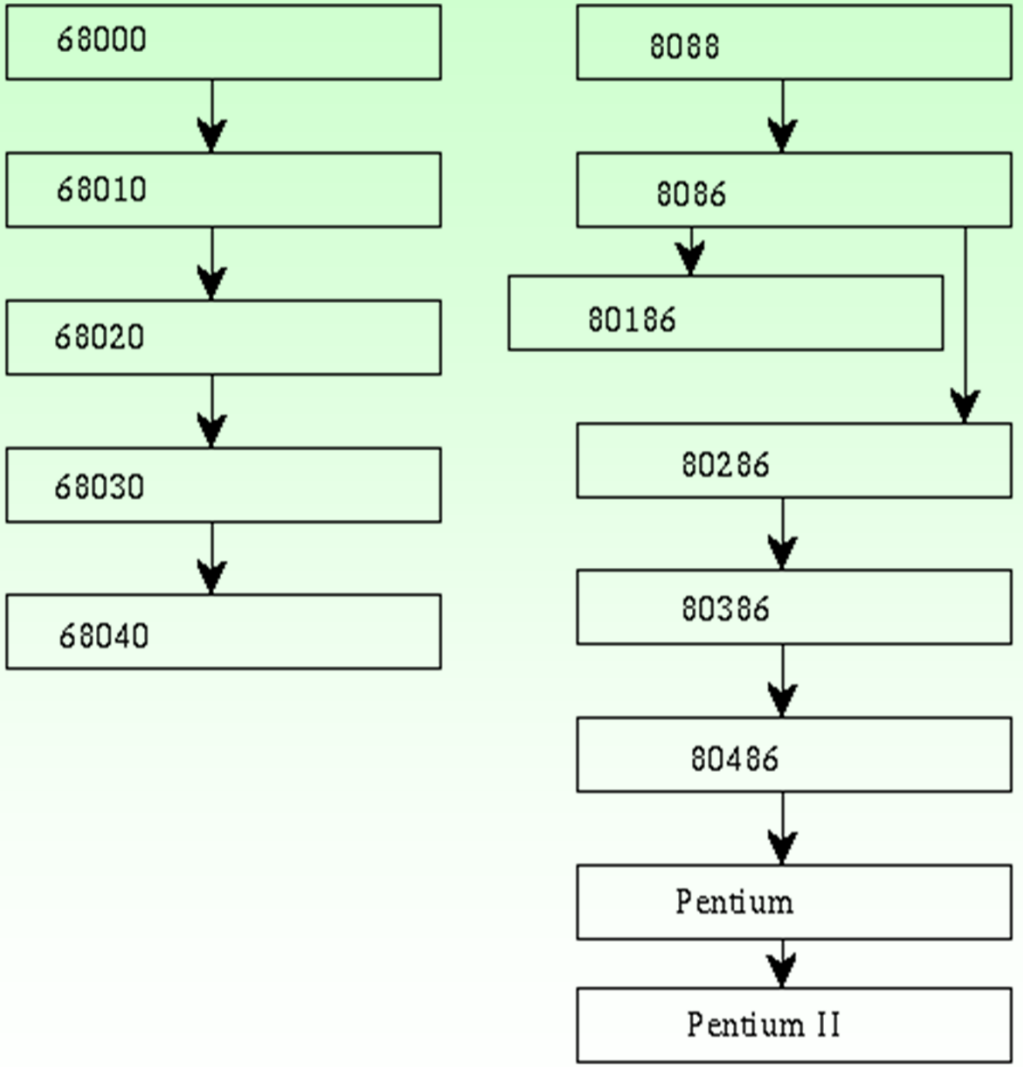
- so that existing programs can use newer and faster processors "as is"
- you can use existing compilers - if you want to!
- you will need to use new compilers to exploit new ISA instructions
  - (that in turn exploit new features of the hardware)
- thus, new CPUs often support all the instructions of the older processor, in addition to some new ones

However, it eventually becomes difficult to maintain compatibility,

- leading to the appearance of new families of architectures, for example: 68000 series and PC series (Motorola).

Families of Processors

- Two major families of microprocessors
- Backwards compatibility is not a new idea
- Motorola now have a new family of processors:
  - PC series.



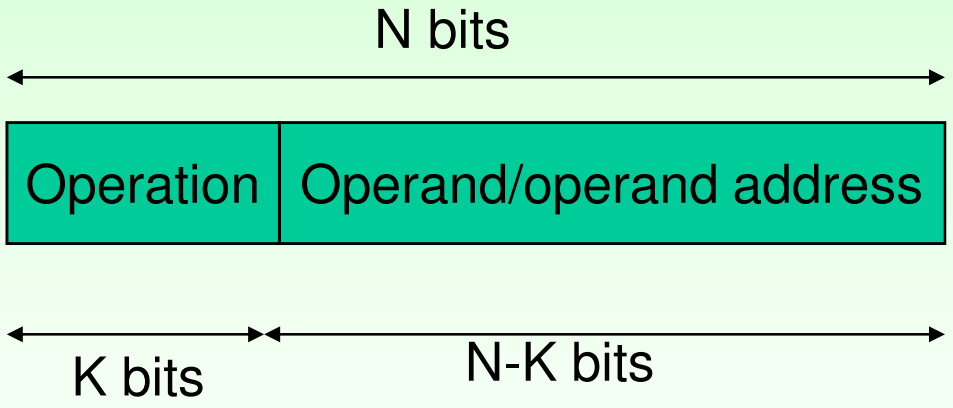
Classes of Instructions

- There are a large number of instructions in the ISA
  - Basically, there are four classes, plus a catch-all miscellaneous class:
- Data Movement
  - moving data from main memory to register, from register to main memory, or from register to register.
- Arithmetic
  - Operating on data items: this includes integer operations, floating point operations, logical operations, and shift operations.
- Tests
  - Testing the values of flags (contents of flag registers), which have been set or unset depending on previous (usually arithmetic) operations.
- Control Transfer
  - Transferring control to some other part of the program, rather than simply executing program instructions sequentially.
- Miscellaneous
  - Other instructions.

Instruction Formats - 1

- Instructions may be stored
  - one per word
  - or over some number of cells

- Simple format



- This allows an N-bit word to code
  - one of 2^K operations and
  - one of $2^{(N-K)}$ operands

Instruction Formats - 2

This can be too simplistic:

- different operations will have different numbers of operands
  - maybe none, maybe as many as three

- One may place an operand or an operand address in a register
  - this allows address arithmetic, which can be useful (e.g.) in array accessing
  - an instruction need not contain an explicit address - it may simply refer to the register where the relevant address is stored.

The precise format used is a compromise between:

1. efficient instruction coding
2. capability to address whole memory efficiently
3. ease of decoding

Number of Operands

- Different instructions have different requirements from their operands.
  - for example there may be 0,1,2, or 3 operands:

    0 Set/Clear a flag, reset, no-operation,...

    1 Clear address, increment register,...

    2 Load a register from address, add two operands, overwriting one with the result, ...

    3 Add two operands, storing the result elsewhere....

- By careful coding, a single fixed-length instruction can accommodate all of these.

- Operands may be registers or main memory locations (addresses), or simply values.

- Operands may be actually present in the instruction, or may be in registers, or may require to be fetched from memory.

3 Operand Instructions

- Apply an operation to two values to compute a third value
  - in this case, the instruction will have separate fields for the two values and the result.
  - generally the word destination is used for the field for the result (the result will be stored somewhere).
  - correspondingly the word source is used for the fields for the values to be used.
  - clearly, three operands could require a large instruction word
    - particularly if some or all of the operands are memory locations
    - but not so much if they are all registers.
- Such instructions are used in RISC (Reduced Instruction Set Computer) processors for arithmetic operations on register data, e.g:

```
ADD R1, R2, R3
```

–> add contents of **R2** and **R3**, and store the result in **R1**

2 Operand Instructions

- These are very widely used.
- For example, LOAD and STORE instructions are generally 2-operand

```
LOAD    R2, Location 1
STORE R3, Location 2
```

- additionally, many computer systems have instructions like

```
ADD  R3, R4
```

- Add the contents of R3 and R4, and store the result in R3.

2 Operand Instructions

When we consider simple arithmetic operations in programs it is the case that statements like

```
xx = xx + yy ;
```

are more common than statements like

```
zz = xx + yy ;
```

2-operand instructions can implement these very efficiently. However 3 operand instructions are more general

```
ADD R1, R1, R2
```

1 Operand Instructions

The task may be to modify a value stored somewhere. In such a case the source and the destination are the same:

```
NEG R1
```

- Negate/flip all the 1's in R1 by 0's, and vice versa

Often an (additional) operand is implicit. For example,

```
CLEAR R0   ; set R0 to 0
```

has an implicit operand of 0, as the value to be placed into R0, or

```
INC R4     ; increment R4
```

has an implicit operand of 1, as the value to be added to R4.

No Operand Instructions

These are used where there really is no operand. e.g.

`NOP`	no operation
`HALT`	stop executing instructions
`RESET`	return the whole CPU to a known state

or where an operand is implicit, e.g.

`CLC`	clear the CARRY flag

and on stack machines ...

Stack Machines

These are sometimes known as 0 operand machines

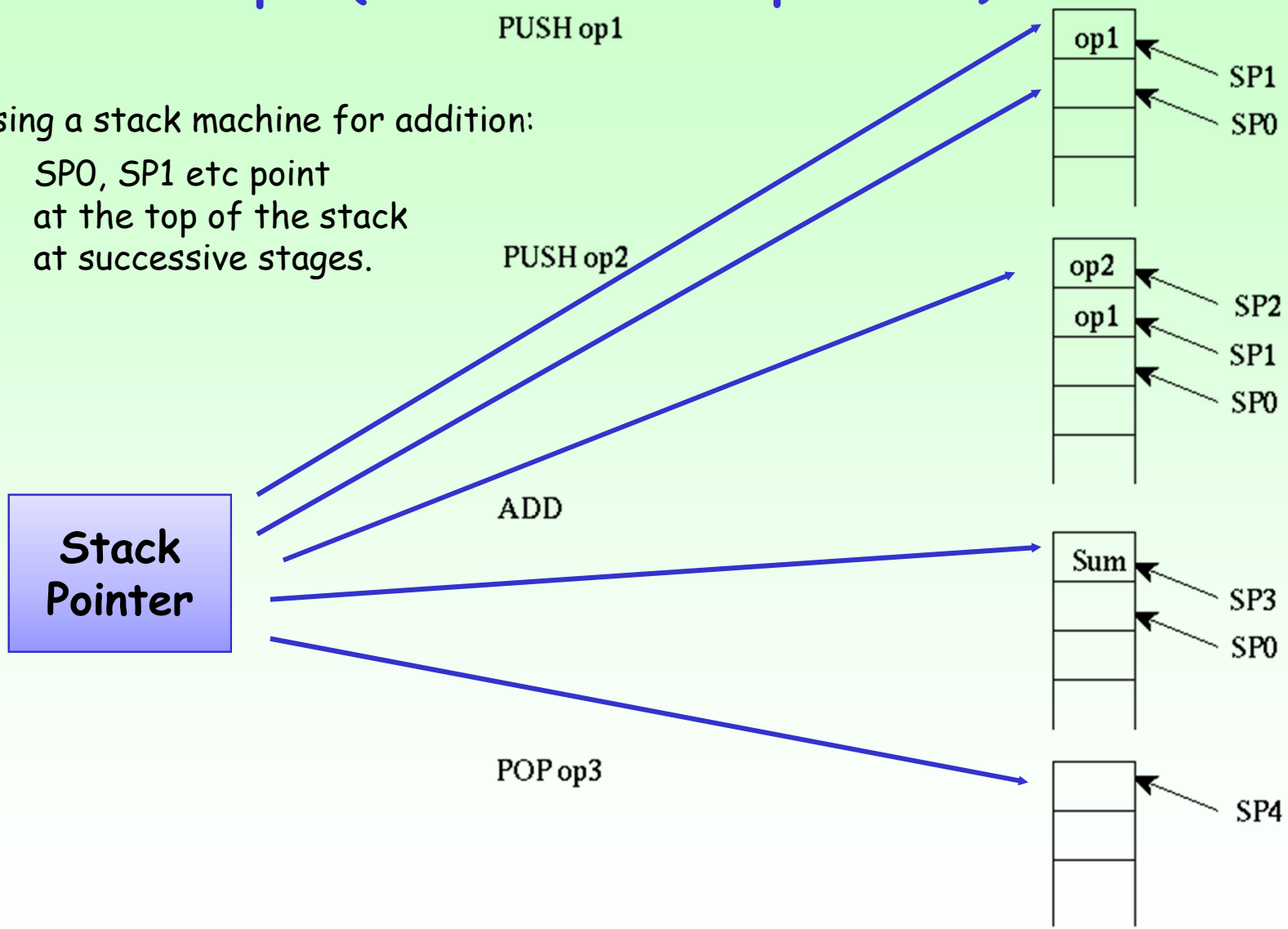
- the machine revolves entirely around a stack.
  - stack
  - last-in first-out store (LIFO)

thus

- all loads are PUSHes (memory content is placed on stack)
- all stores are POPs (top stack element is written to memory)
- all arithmetic uses (and replaces) either:
  - the top-of-stack item (for a unary operation, like negate)
  - the top two items on the stack (for a binary operation, like add or subtract)

example (stack machine operation)

- Using a stack machine for addition:
  - SP0, SP1 etc point at the top of the stack at successive stages.



Reverse Polish Notation

- It is straightforward to parse arithmetic instructions into a form where they can be used on a stack machine:
  - c = a + b

    becomes PUSH a; PUSH b; +; POP c;

  - y = (x + y) * z

    becomes PUSH x; PUSH y; +; PUSH z; *; POP y;

- This way of writing an arithmetic expression is known as Reverse Polish or Łukasiewicz notation.

- It is used exclusively on some processors (specifically the transputer series), and is usable on many other processors,
  - even if they were not designed with this in mind.

Addressing Modes

- There are many different ways of specifying the operands (sources and destination) of an instruction:
  - the *operand* may be an explicit value contained in the instruction,
  - the *operand* may be in a specified register,
  - the *address of the operand* may be specified in the instruction,
  - the *address of the operand* may be in a specified register,
  - the *address of the operand* may be at a location whose address is contained in the instruction,
  - etc.

- In general, the instruction must contain an indication of either what the operand is or how to find it. The way this is specified is known as the *addressing mode*.
- The reference to the operand has to include information about which addressing mode is to be used.
- This is another aspect of the structure of instructions.

Immediate Addressing

The operand is explicitly given in the program code, for example:

```
LOAD  R1,  #5       ; load register 1 with the value 5
ADD   R5,  #2       ; add 2 to the value in register 2
```

the immediate operand is the number (#5 or #2).

Different machines take different attitudes to the coding of this.

- for small values (say <256) the value may be in the instruction word.
- for larger values the value may be in the word following the instruction.

Immediate addressing is often used:

- for adding small values to registers
  - e.g. incrementing values, incrementing pointers
- for values which occur precisely once in a program
  - y = 6543214 ;
- (if values occur more than once, better techniques exist).

Register Addressing

We saw already the case where the operand is already in the register, or where the result is to be placed in a register.

- this is known as register direct addressing.
- it is useful to place operands in registers, as the instructions are shorter.

One need only specify which register the operand is in

- for 16 registers, this takes 4 bits
- for 32 registers, this takes 5 bits, etc.

Many instructions receive their operands from registers, and place their results in registers:

```
ADD   R4,R5          ; R4 = R4 + R5
SUB   R1,R2,R3                    ; R1 = R2 - R3
LOAD R9,R1                  ; R9 = R1
```

Of course, at some point, we also need to move data from memory to register, and back again.

Direct Addressing

- Direct addressing is one technique for accessing operands which are in the main memory.
  - in direct addressing, the address of the operand is in the program code.
  - Example:

```
STORE R1,13345        ; store R1 at location 13345
```

(In assembly language one would normally have a symbol representing 13345, rather than the number itself.)

- Such instructions need to be more than one word long, as the address needs to be held in the program code.
  - Indeed, the address itself would need to be 32 bits long, for a 32-bit address machine.

Problems with Direct Addressing

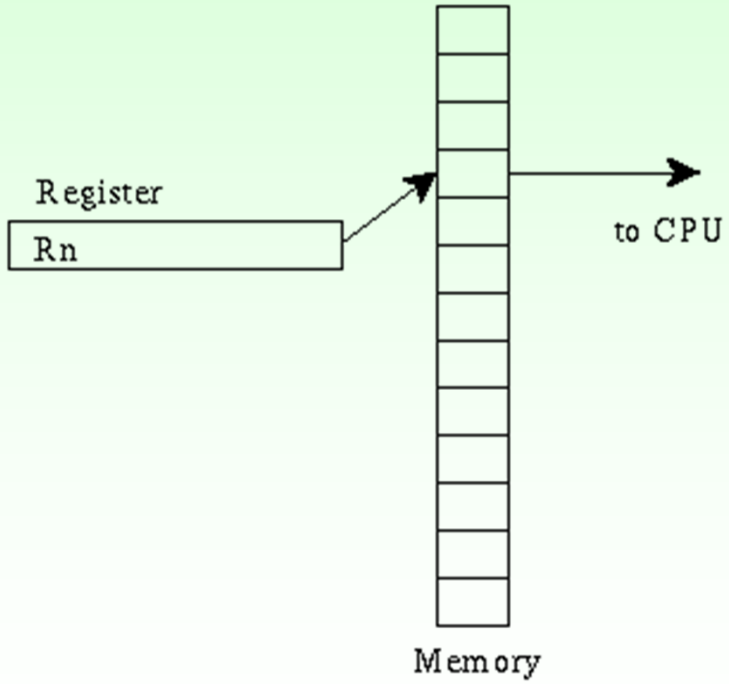
- The major problem with direct addressing is that the addresses are fixed at compilation.
  - the instruction always refers to the same operand address.

- This mode is useful only for accessing global variables.

- Other kinds of stored values may cause difficulty:
  - local variables in methods,
  - array elements accessed successively (in a loop)
  - any variables whose storage is determined dynamically (at run-time rather than at compile-time).

Register Indirect Addressing

- The address of the operand is held in a register
  - Allows for computation of the Effective Address (EA) by doing arithmetic and storing result in the register



Why use Register Indirect Addressing?

- This form of addressing allows the same instruction to be applied over and over again to different operands (or to store results in different places) each time.
  - This is clearly useful in array accessing, for example

- For the simple C++ program

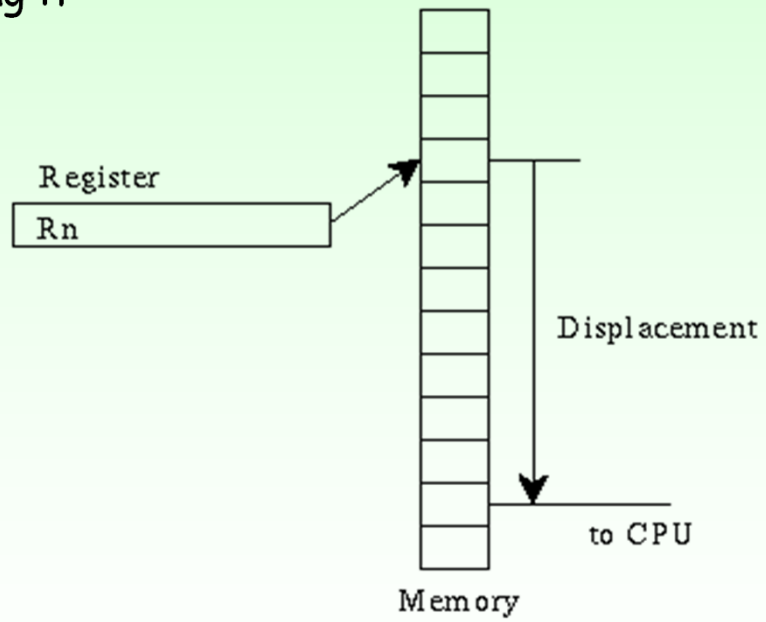
```
int xyz[100] ;
for (int i=0;i<100;i++)
   xyz[i] = i;
```

- One might place the address of the array in (e.g.) register R4, then use register addressing to store the value of i using the same instruction every time incrementing the value in R4 each time

Variants of Register Indirect Addressing

Many variants exist - these range through:

- predecrement
  - decrement the register before accessing it
- postincrement
  - increment the register after accessing it
- displacement
  - The operand address is calculated by adding the displacement on to the value in the address register.



Instructions Concluded

- Many older computer systems have huge numbers of instructions
  - Specialised instructions for specific tasks
  - Many different ways of specifying operands
- But many modern machines have a much smaller number
  - Don't add new instructions for specialised tasks
  - Instead make the smaller number of instructions very efficient
  - Use only direct, register and register indirect addressing modes
- These are known as Reduced Instruction Set Computers (or RISC)
- Most modern processors are RISC internally (Core Duo, P4, G5 etc.)

Instructions Concluded

- Reduced Instruction Set:

```
LOAD R1, 0x52
LOAD R2, 0X74
PROD R1, R2
STORE R1, 0x52
```

- Complex Instruction Set:

```
MULT  0x52, 0x74
```

Which do you choose?

$$\frac{time}{program} = \frac{time}{cycle} \times \frac{cycles}{instruction} \times \frac{instructions}{program}$$

End of Lecture