

Chapter 2: Computer Organisation

- A simple computer and its instruction set
- Communicating with devices
- Introduction to interrupts

Olav Beckmann

Huxley 449

<http://www.doc.ic.ac.uk/~ob3>

Acknowledgements: There are lots. See end of Chapter 1.

Home Page for the course:

<http://www.doc.ic.ac.uk/~ob3/Teaching/OperatingSystems/>

This is only up-to-date after I have issued printed version of the notes, tutorials, solutions etc.

Computer Organisation: Chapter Overview

- How does a computer work?
- Enough to understand key OS ideas:
 - how instructions are executed
 - how can a processor send and receive data from another device
- Textbook
 - Tanenbaum Chapter
 - Nutt Chapter 4

The NARC Instruction Set

<i>Instruction opcode</i>		<i>Meaning</i>
LOADC	1	ACC:= D
LOADM	2	ACC:= Memory(D)
STOREM	3	Memory(D):=ACC
ADDC	4	ACC:=ACC + D
ADDM	5	ACC:=ACC + Memory(D)
SUBC	6	ACC:=ACC - D
SUBM	7	ACC:=ACC - Memory(D)
JMP	8	PC:=D
JMPZ	9	If ACC=0 then PC:=D
JMPN	10	If ACC<0 then PC:=D
CALL	11	LNK:=PC; PC:=D
RET	12	PC:=LNK
HALT	13	

What exactly does the processor do?

```
int Mem[MAX]; // main memory
```

```
// Internal registers of the processor
```

```
int Acc; // accumulator
```

```
int Lnk; // link register
```

```
int PC; // program counter
```

```
int Op; // current opcode
```

```
int D; // current operand
```

```
void fetch() {
```

```
int W;
```

```
W = Mem[PC];
```

```
Op = Opcode(W); // most significant 8 bits of W
```

```
D = Operand(W); // least significant 24 bits of W
```

```
}
```

*This piece of C
code describes
what the
processor does to
fetch an
instruction*

Execute

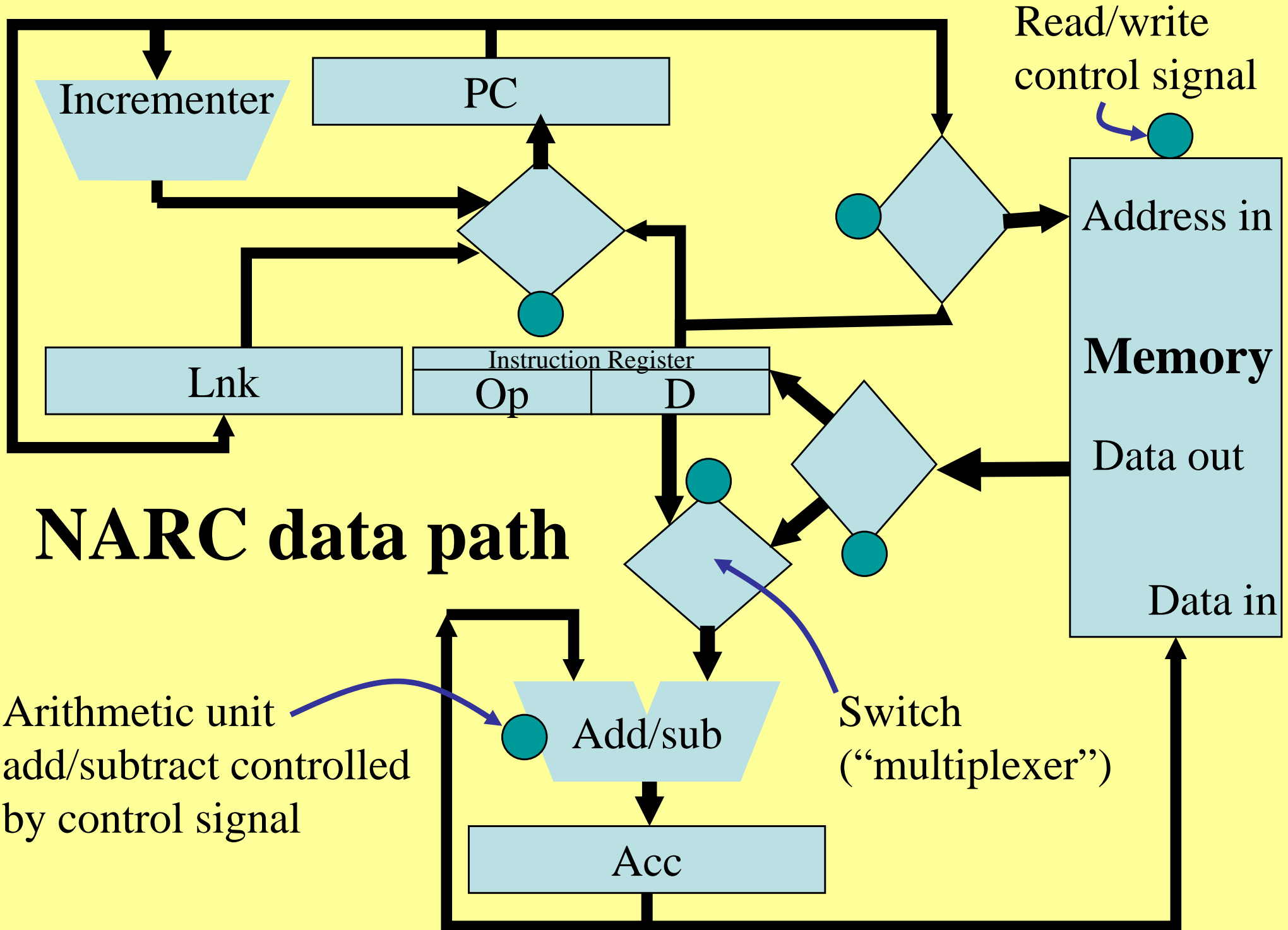
```
void execute() {  
    switch(op) {  
        case 1:  Acc = D;           // (opcode)  
                // loadc  
        case 2:  Acc = Mem[D];      // loadm  
                // loadm  
        case 3:  Mem(D) := Acc;     // storem  
                // storem  
        case 4:  Acc = Acc+D;       // addc  
                // addc  
        case 5:  Acc = Acc+Mem[D];  // addm  
                // addm  
        case 6:  Acc = Acc-D;       // subc  
                // subc  
        case 7:  Acc = Acc-Mem[D];  // subm  
                // subm  
        case 8:  PC = D;            // jmp  
                // jmp  
        case 9:  if (Acc=0) PC:=D;  // jmpz  
                // jmpz  
        case 10: if (Acc<0) PC:=D;  // jmpn  
                // jmpn  
        case 11: Lnk=PC; PC=D;      // call  
                // call  
        case 12: PC=Lnk             // ret  
                // ret  
    }  
}
```

This C function describes what the processor does to execute an instruction

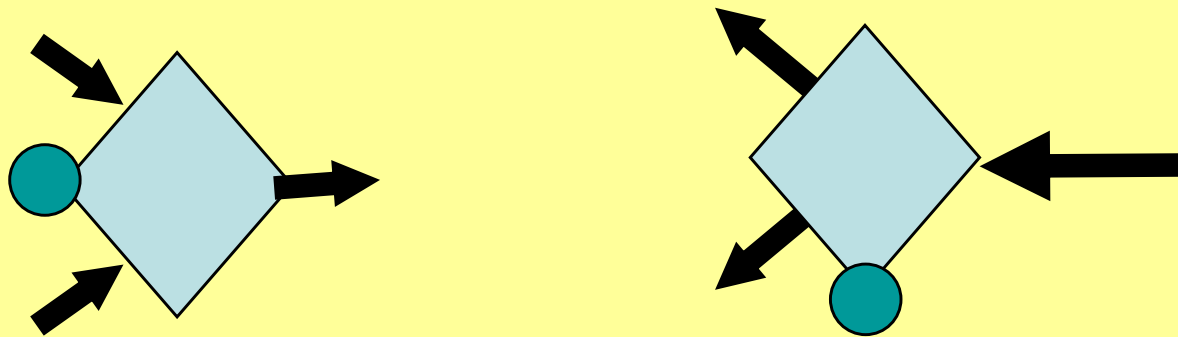
The "Fetch-Execute Cycle"

```
PC = 0;  
do {  
    fetch( );  
    PC=PC+1;  
    execute( );  
} forever;
```

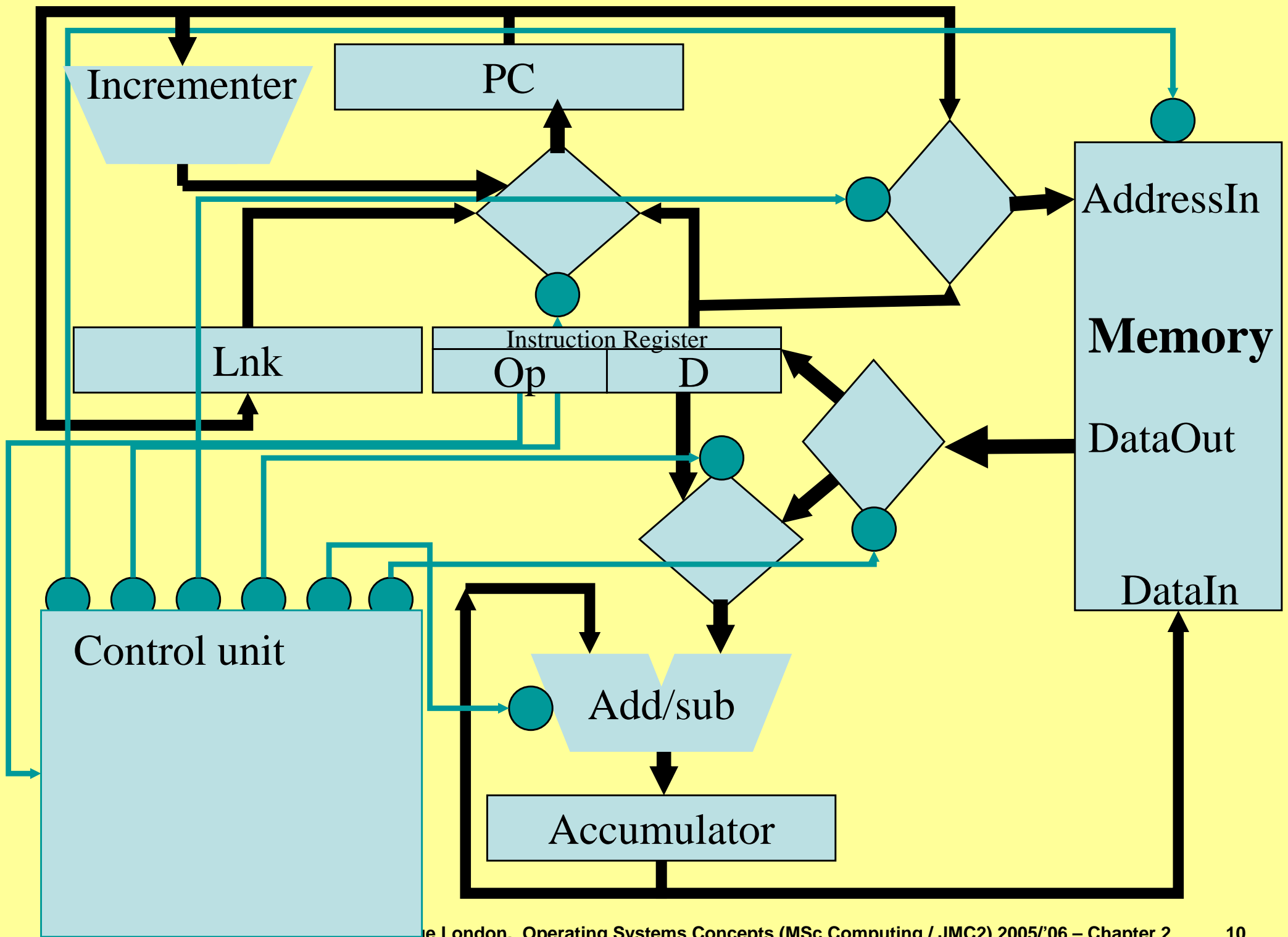
- Execution starts at location zero (when NARC boots)
- Instructions are stored in memory
- So are the computation's working variables
- Fetch and execute are realised as digital circuits (see next slide)



- The data path shows how data flows through the machine as instructions are executed
- The flow of data is controlled by switches (sometimes called *multiplexers/demultiplexers*):



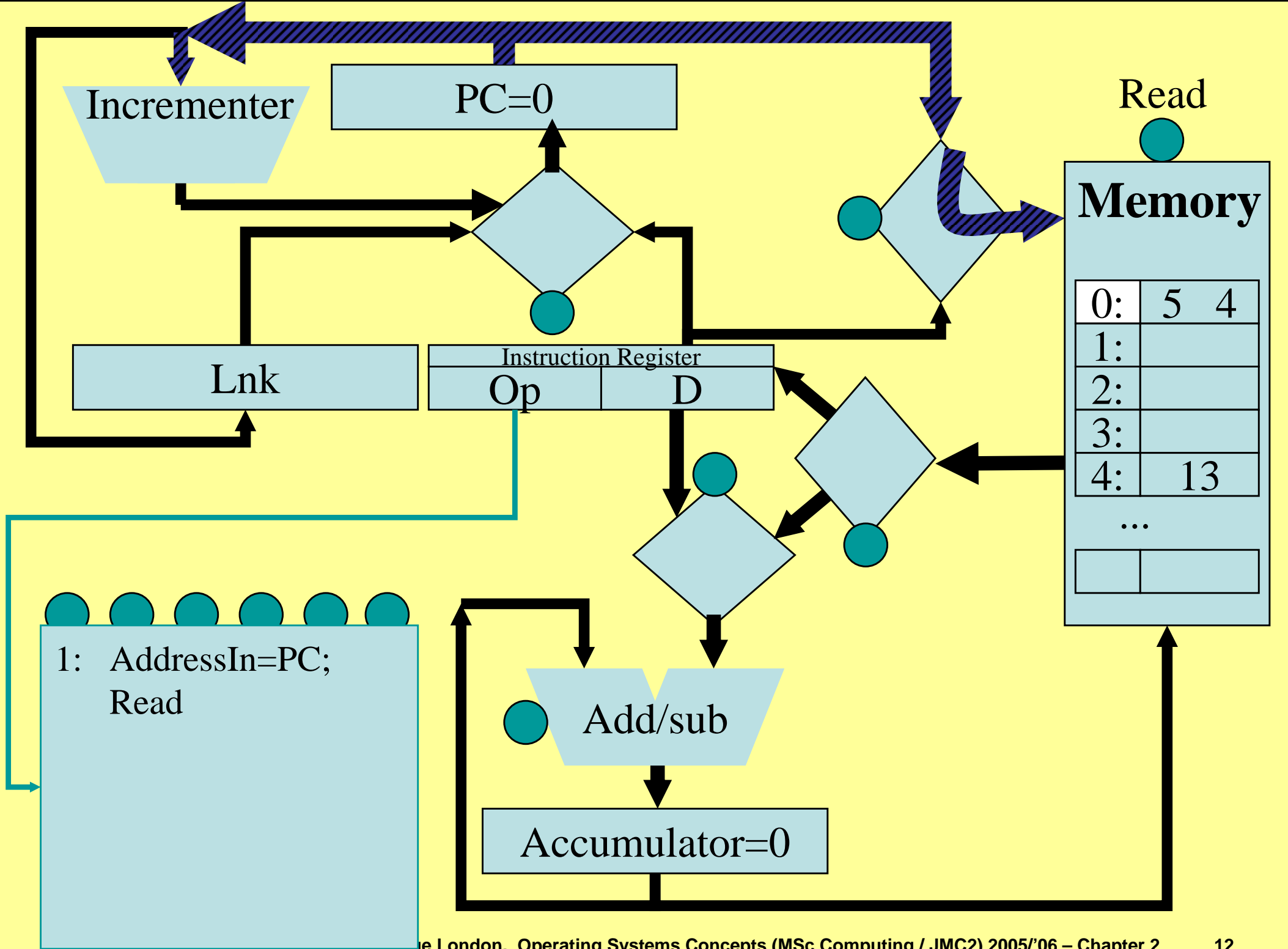
- The arithmetic unit can add or subtract
- The memory can read or write
- In each case, what happens to the data is determined by a control signal ●

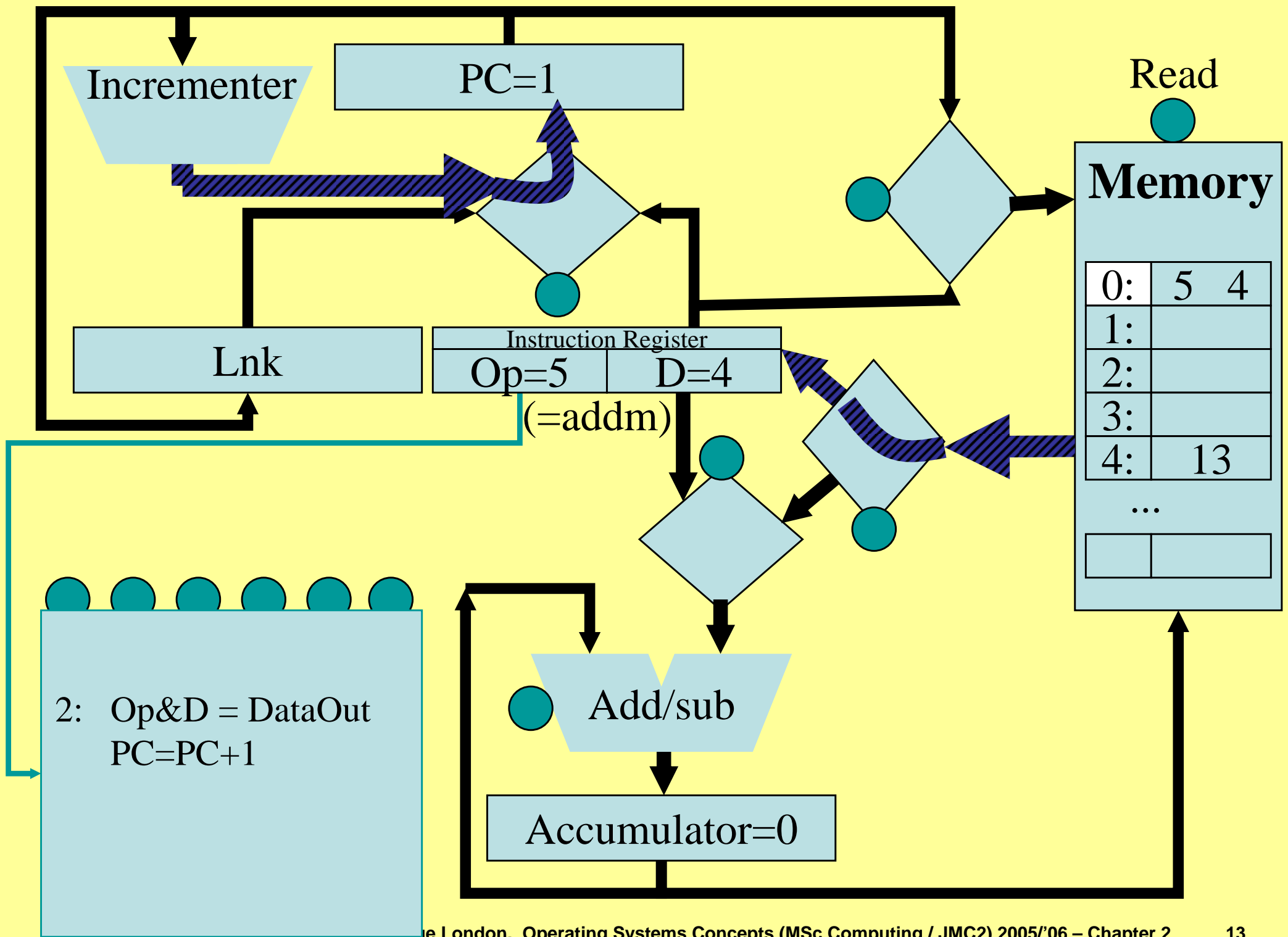


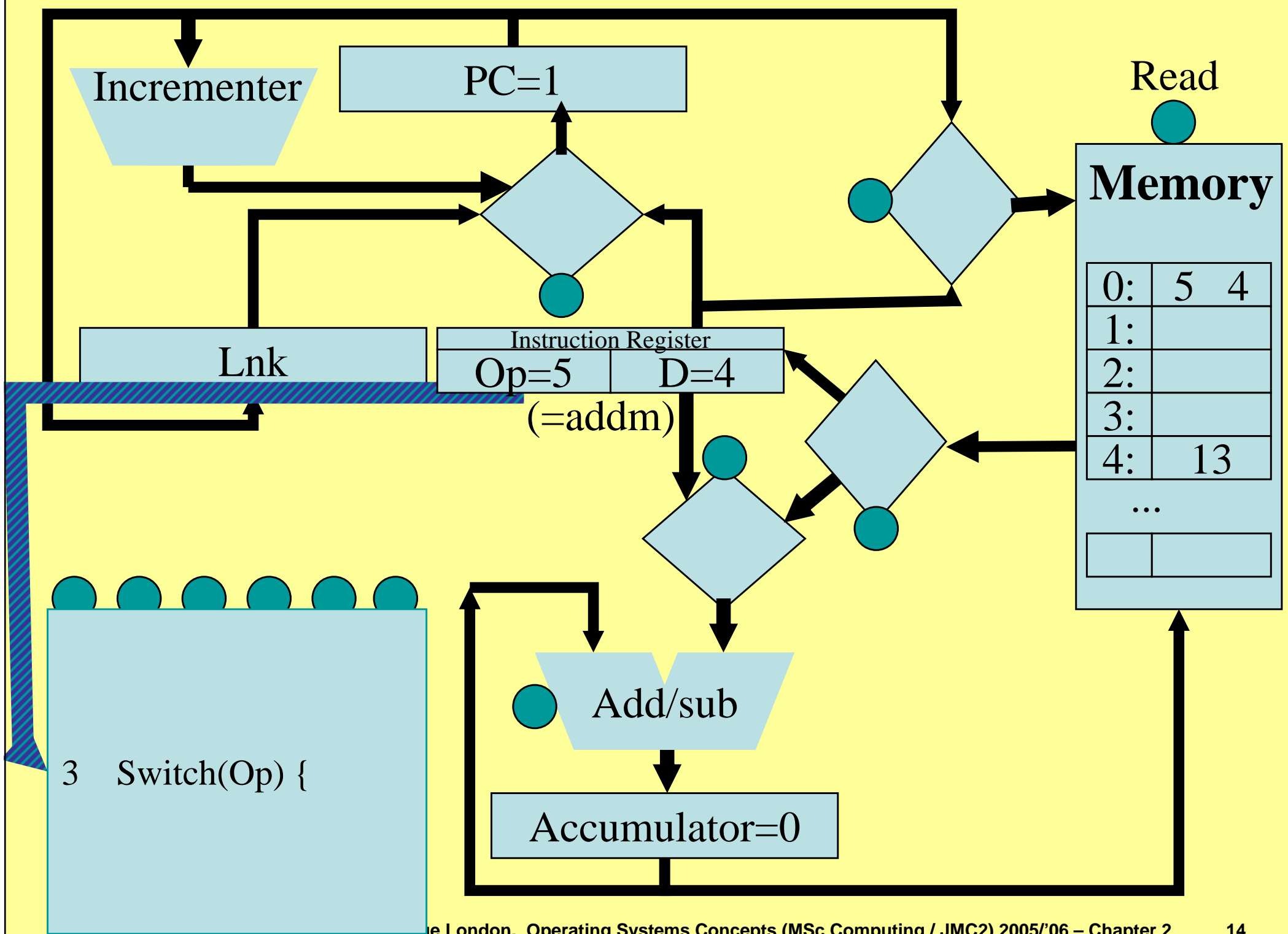
Instruction execution

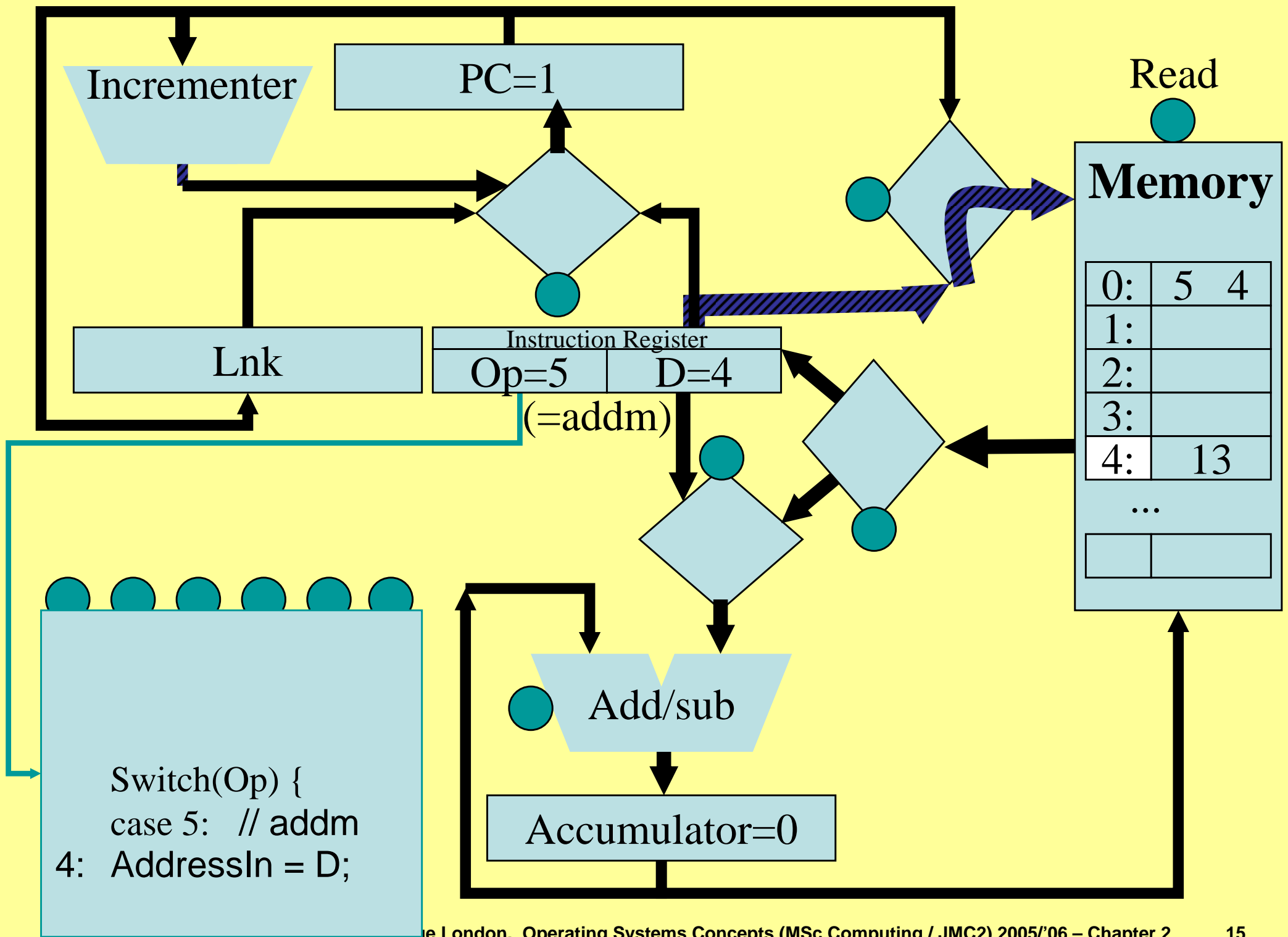
- The control unit generates a sequence of control signals
- First, the PC (Program Counter) is passed to the Memory as its Address In signal
- The Memory is set to “Read”
- The DataOut is passed to the Instruction Register (which consists of two parts: OpCode and Data)
- Meanwhile the PC is passed to the incrementer so that it is ready to point to the next instruction

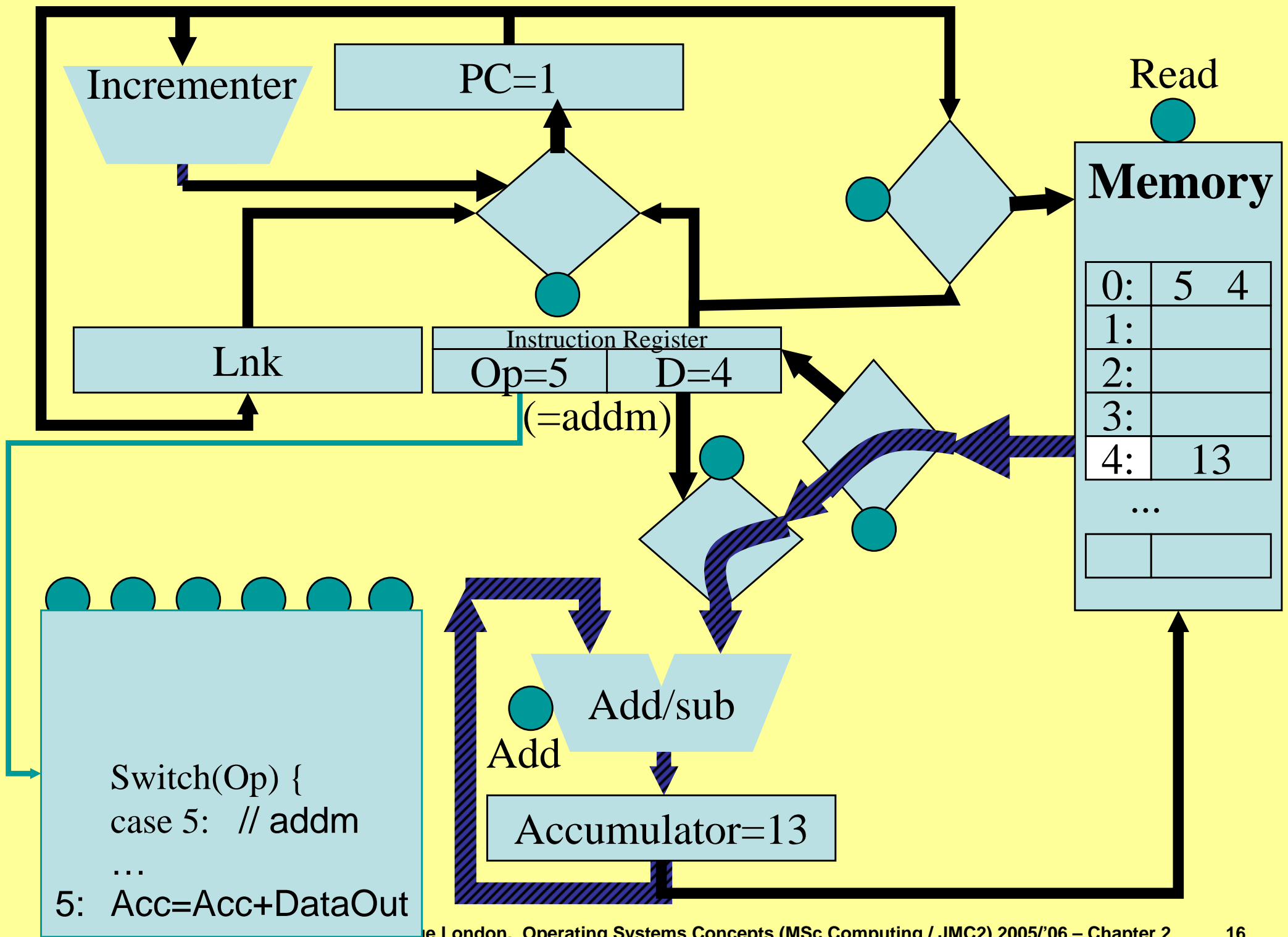
Example: “addm”...







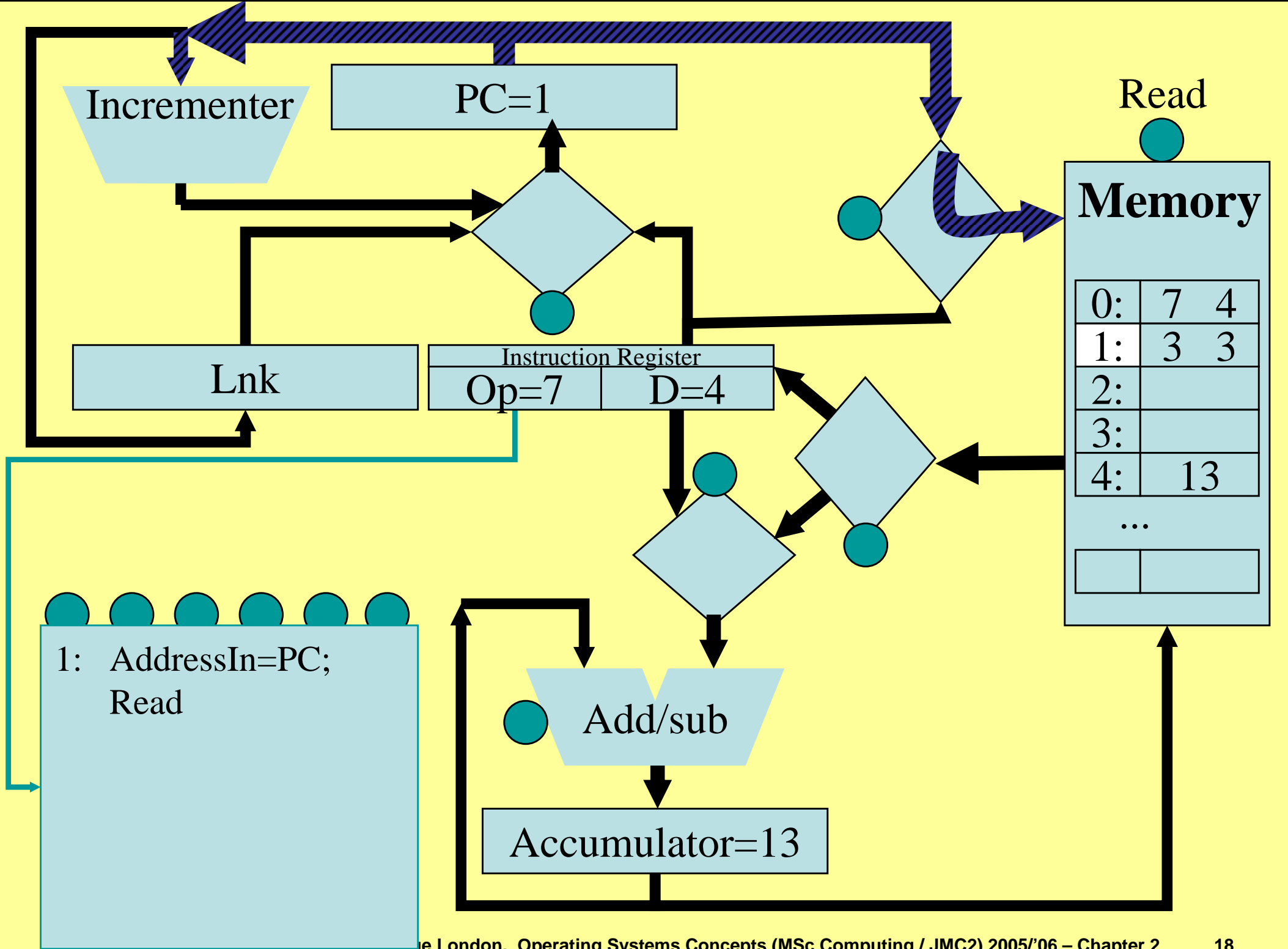


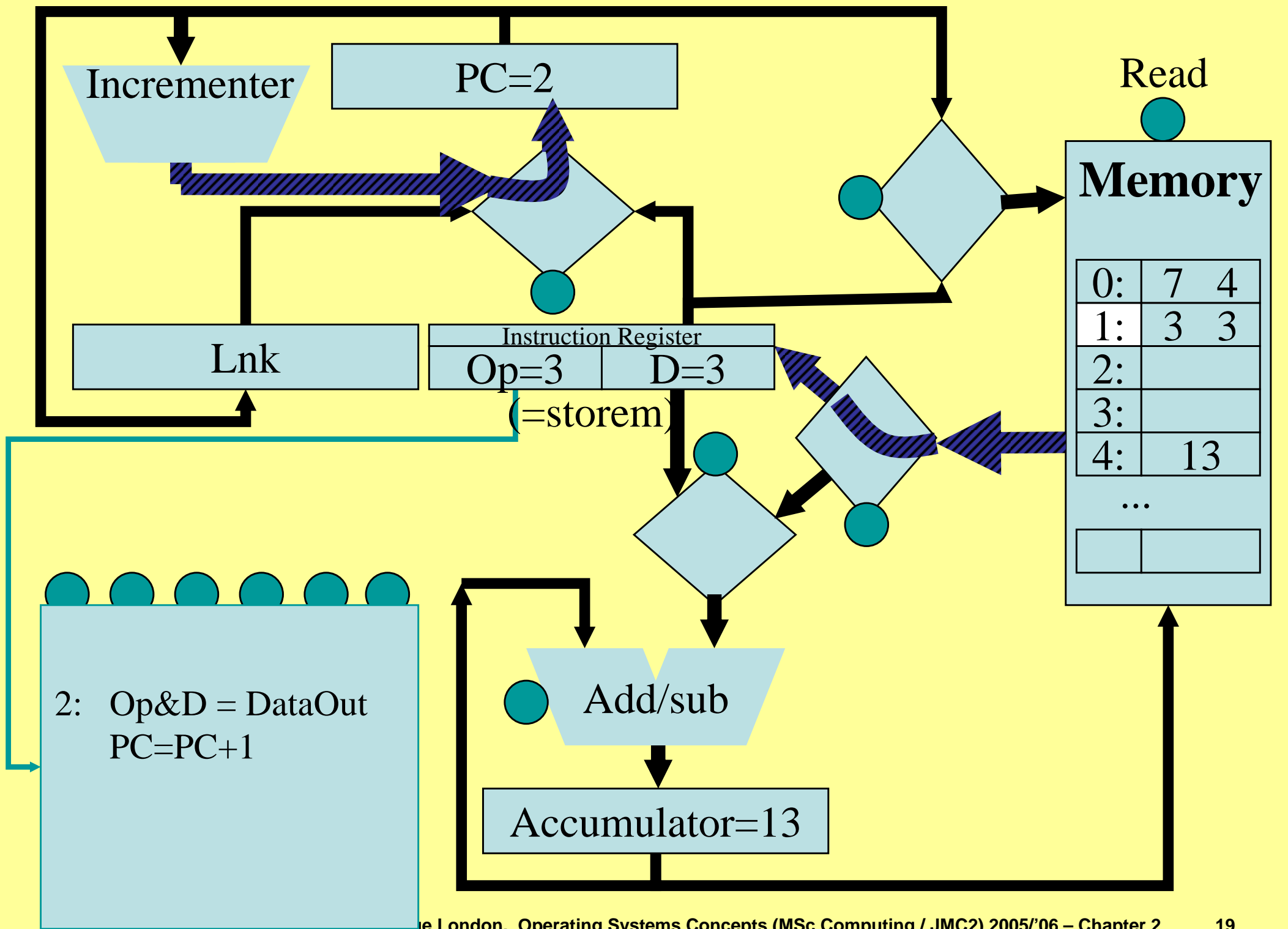


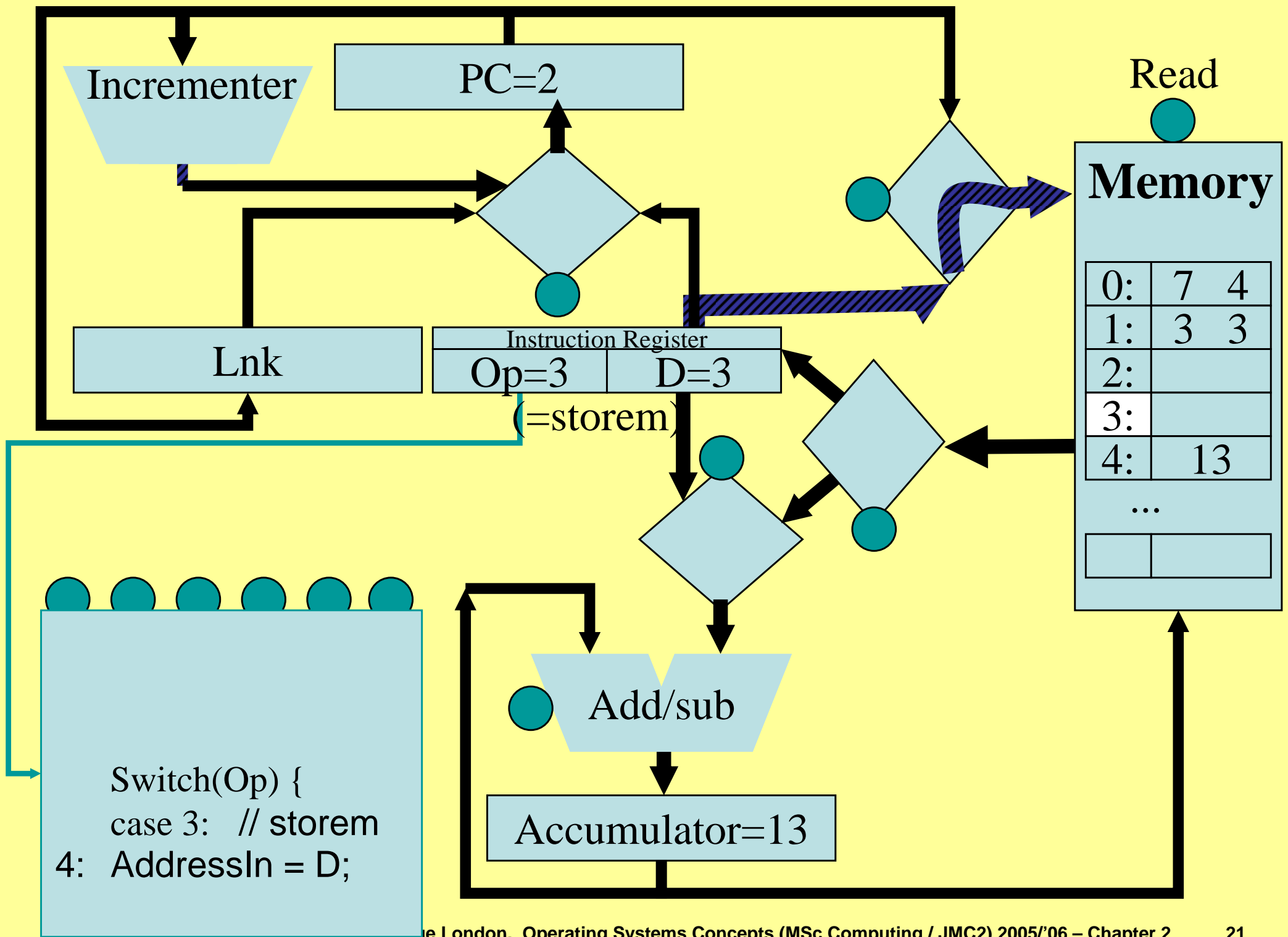
Exercise

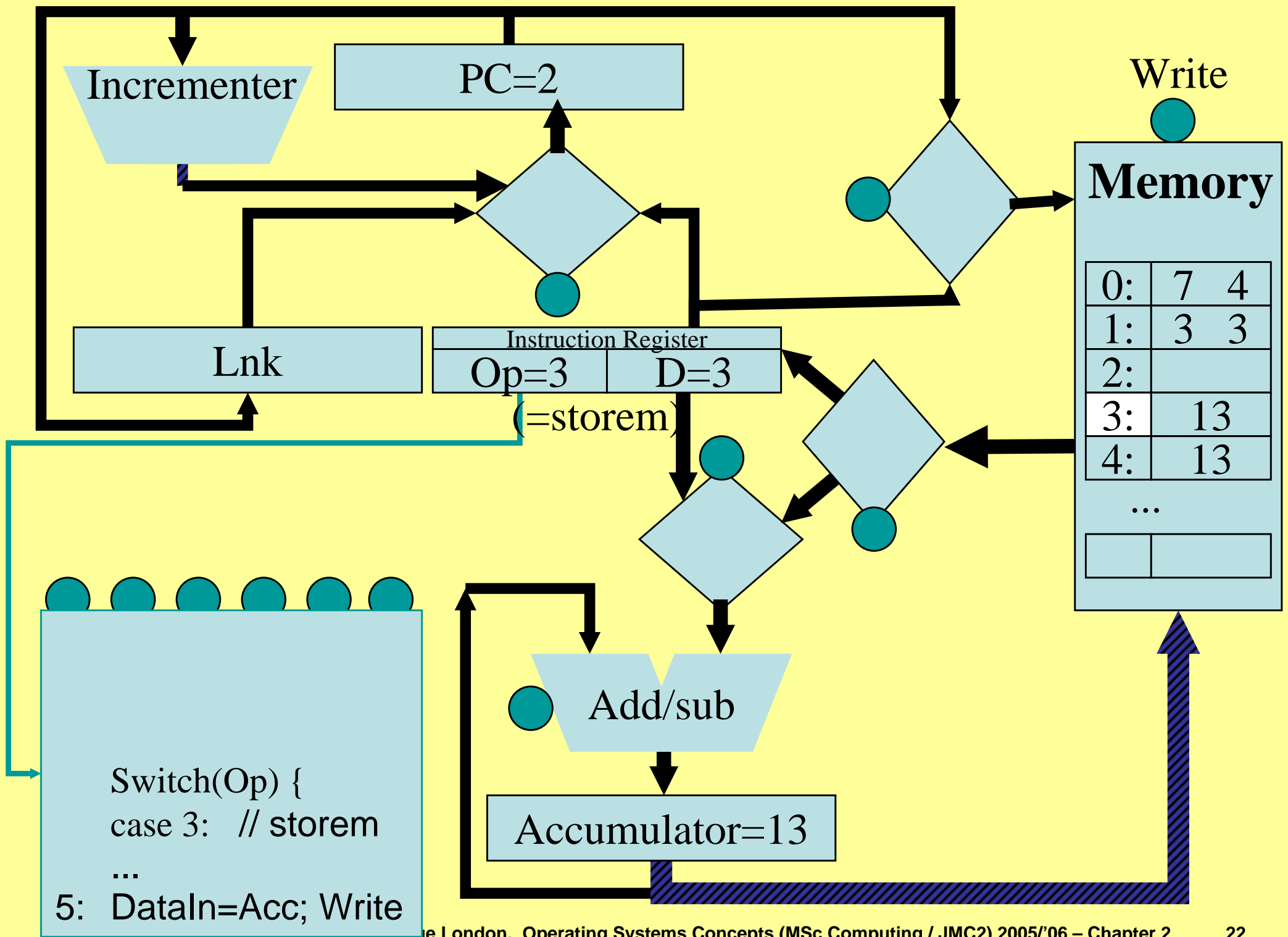
- Suppose that word 2 of the memory contains a “storem 3” instruction:
case 3: Mem(D) := Acc; // storem
- Trace through the sequence of operations to execute this instruction

Memory		
0:	5	4
1:	3	3
2:		
3:		
4:	13	
...		





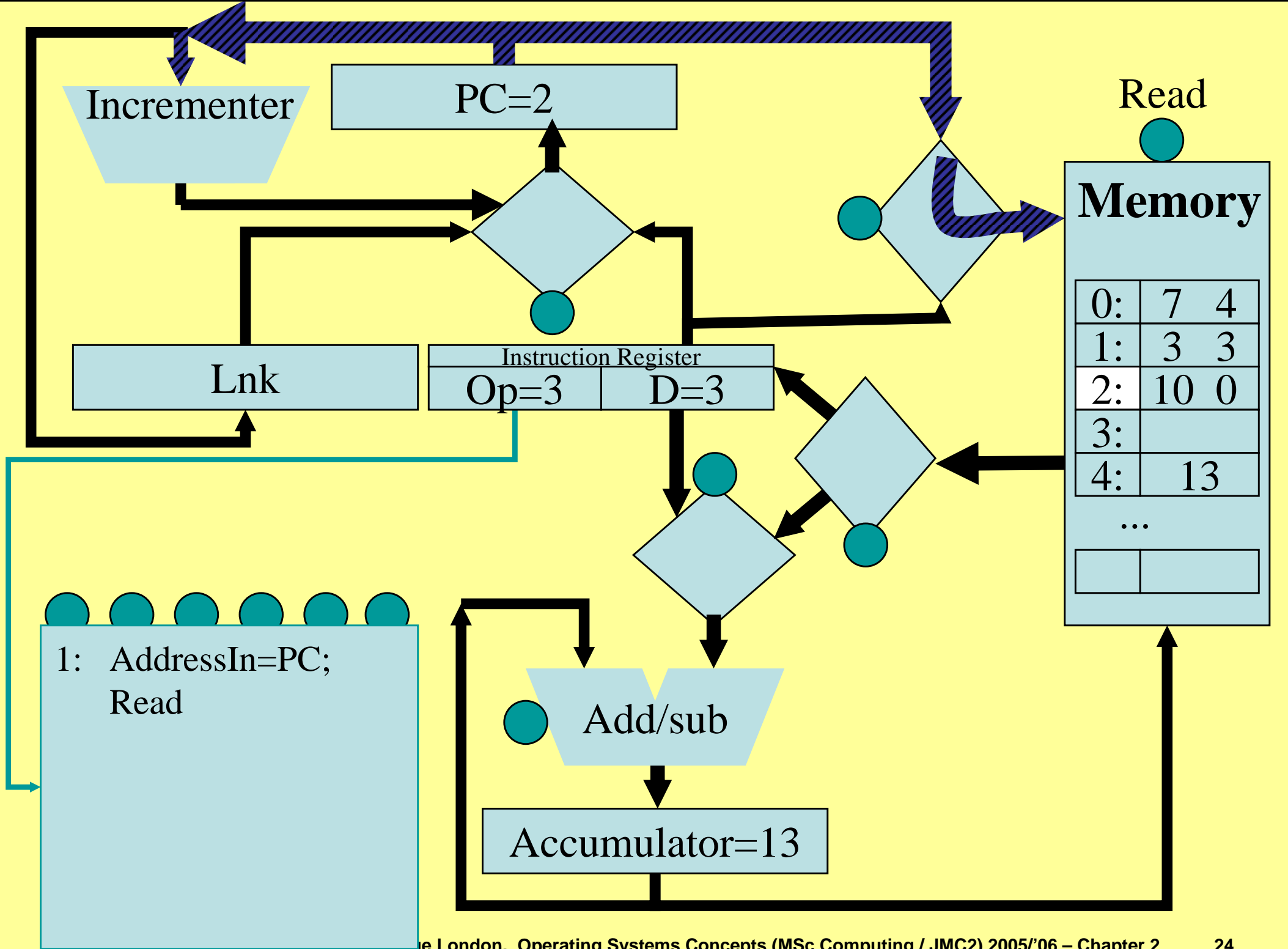


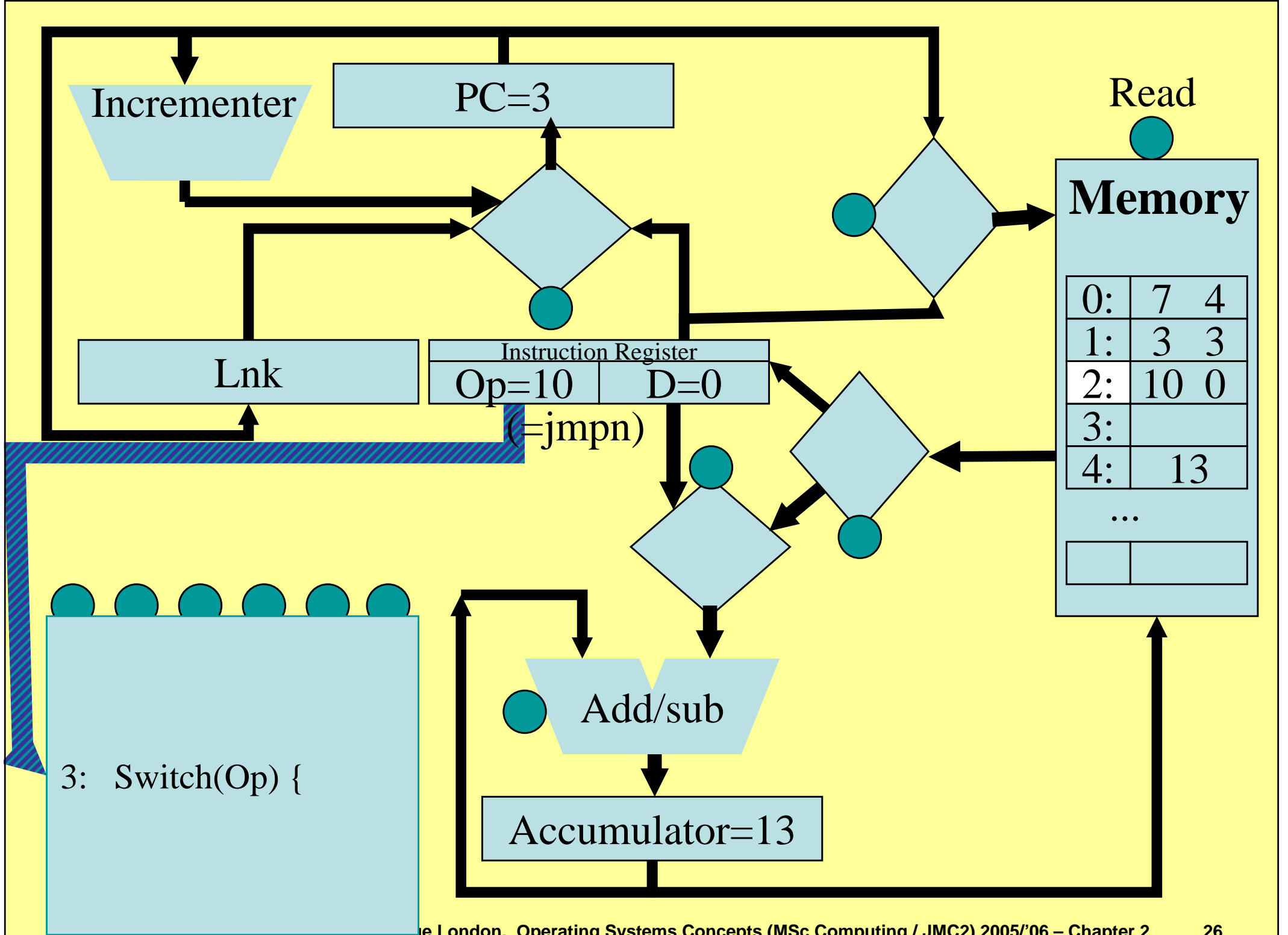


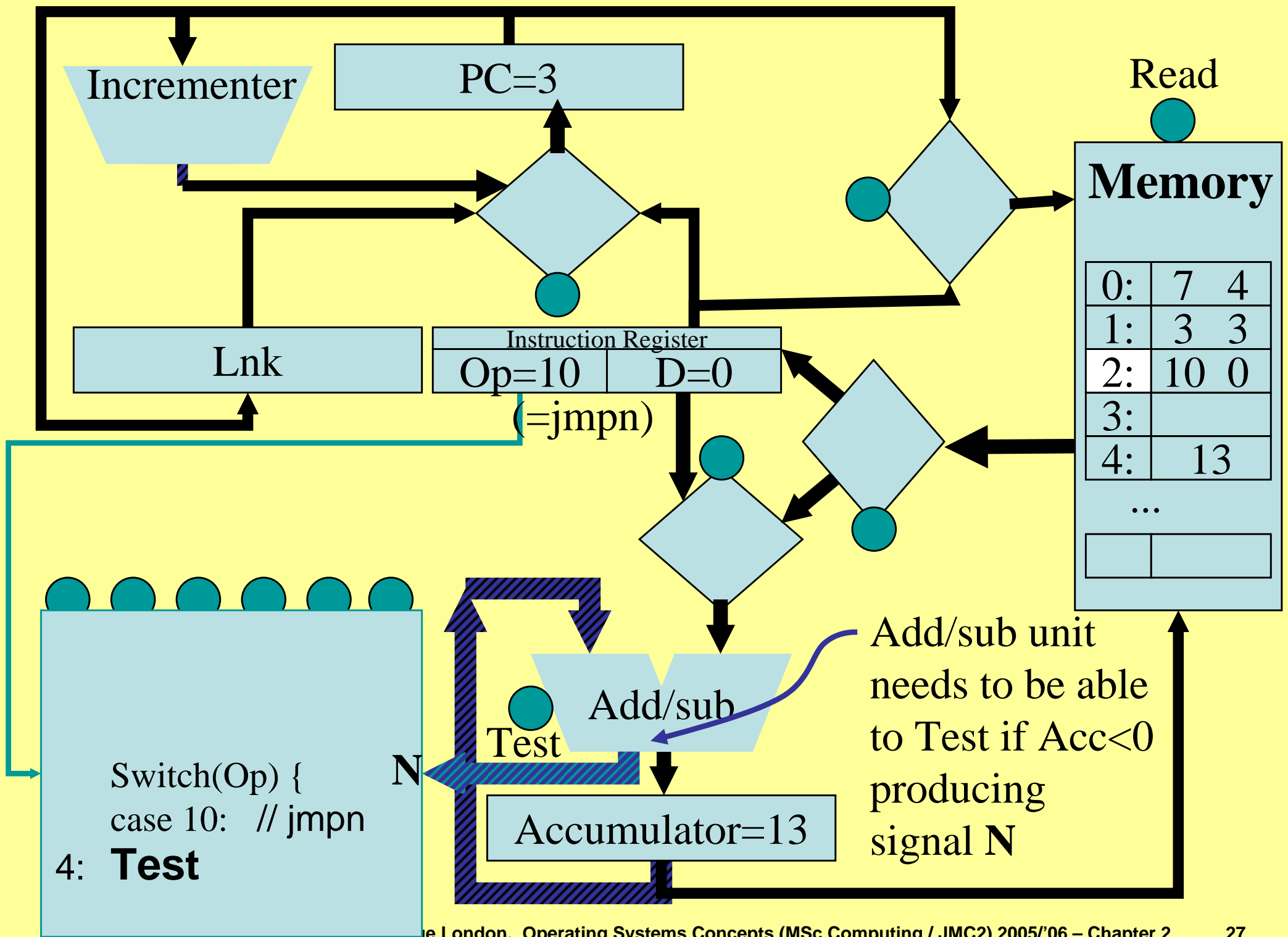
Exercise

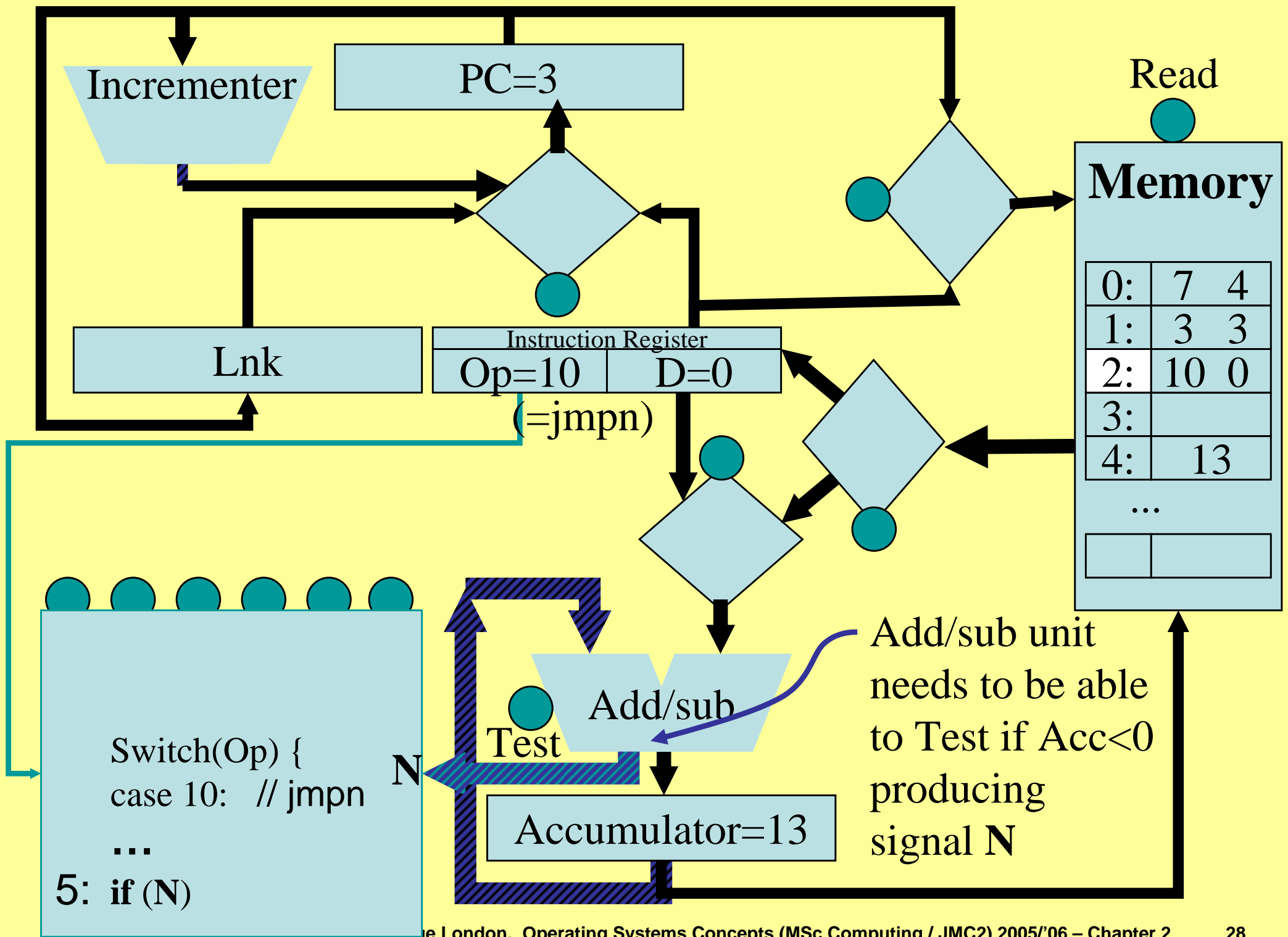
- Suppose that word 3 of the memory contains a “jmpn 0” instruction:
case 10: if (Acc<0) PC:=D; // jmpn
- Trace through the sequence of operations to execute this instruction
- You need to add something to the data path!

Memory	
0:	5 4
1:	3 3
2:	10 0
3:	
4:	13
...	

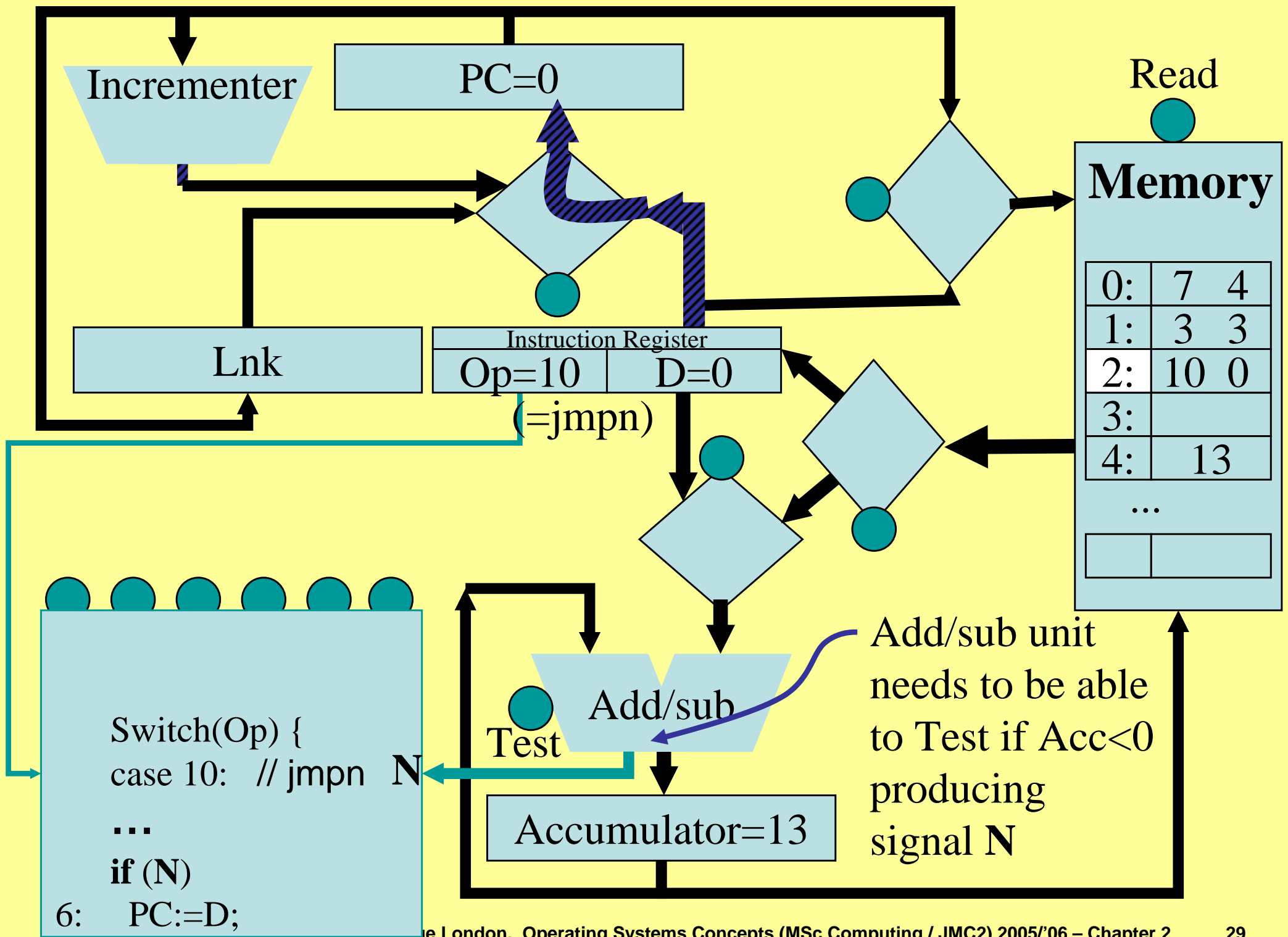








Add/sub unit needs to be able to Test if $Acc < 0$ producing signal N



NARC Datapath - Summary

- The fetch-execute cycle
- Instructions and data are all stored in the same memory
- So one program can operate on another

This is the famous “von Neumann” principle - the concept of a stored-program computer (EDSAC, ca. 1946)

NARC - Control

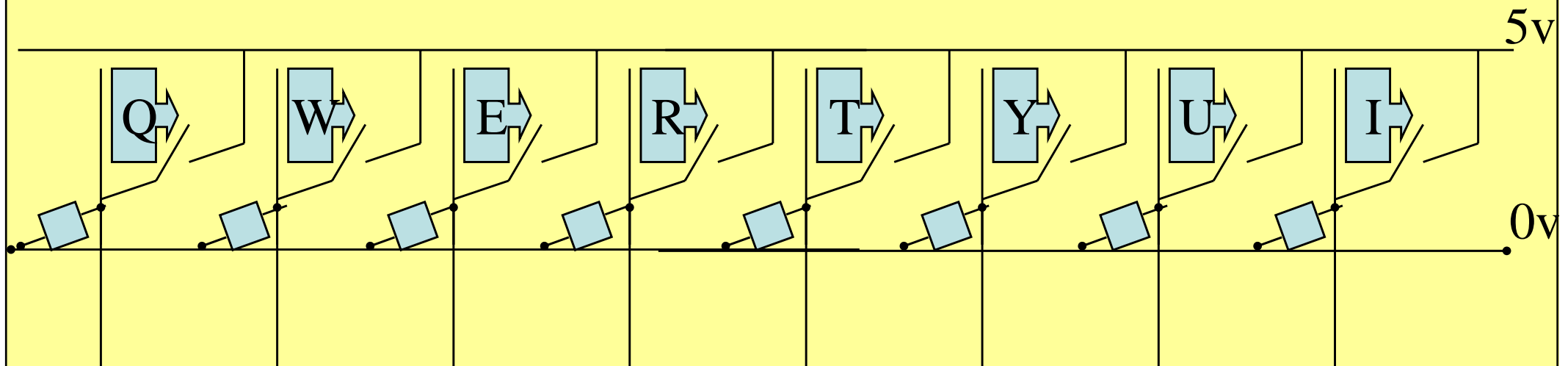
- Each instruction in the NARC instruction set is implemented by a sequence of steps in which data is moved around the data path
- Eg. addm D:
 - 1: AddressIn=PC; Read
 - 2: Op&D = DataOut; PC=PC+1
 - 3 Switch(Op) {
 - 4: AddressIn = D;
 - 5: Acc=Acc+DataOut
- These steps are sometimes called “microinstructions”

NARC is much simplified

- NARC is a simplified computer architecture; realistic architectures work in basically the same way, but
 - instead of just one accumulator, have many registers
 - have multiple arithmetic units that can do multiplication etc
 - are pipelined: while one instruction is being executed, the next is being fetched
 - are very pipelined: while one instruction is using arithmetic unit, the next is accessing registers, the next is being fetched, etc

Turning a Keypress into a Signal

- Suppose we need to connect a keyboard to the NARC
- Each key makes an electrical contact
- Each key generates a signal - 1/0



- Now, how can we connect this to the NARC?

Connecting a keyboard... the memory bus

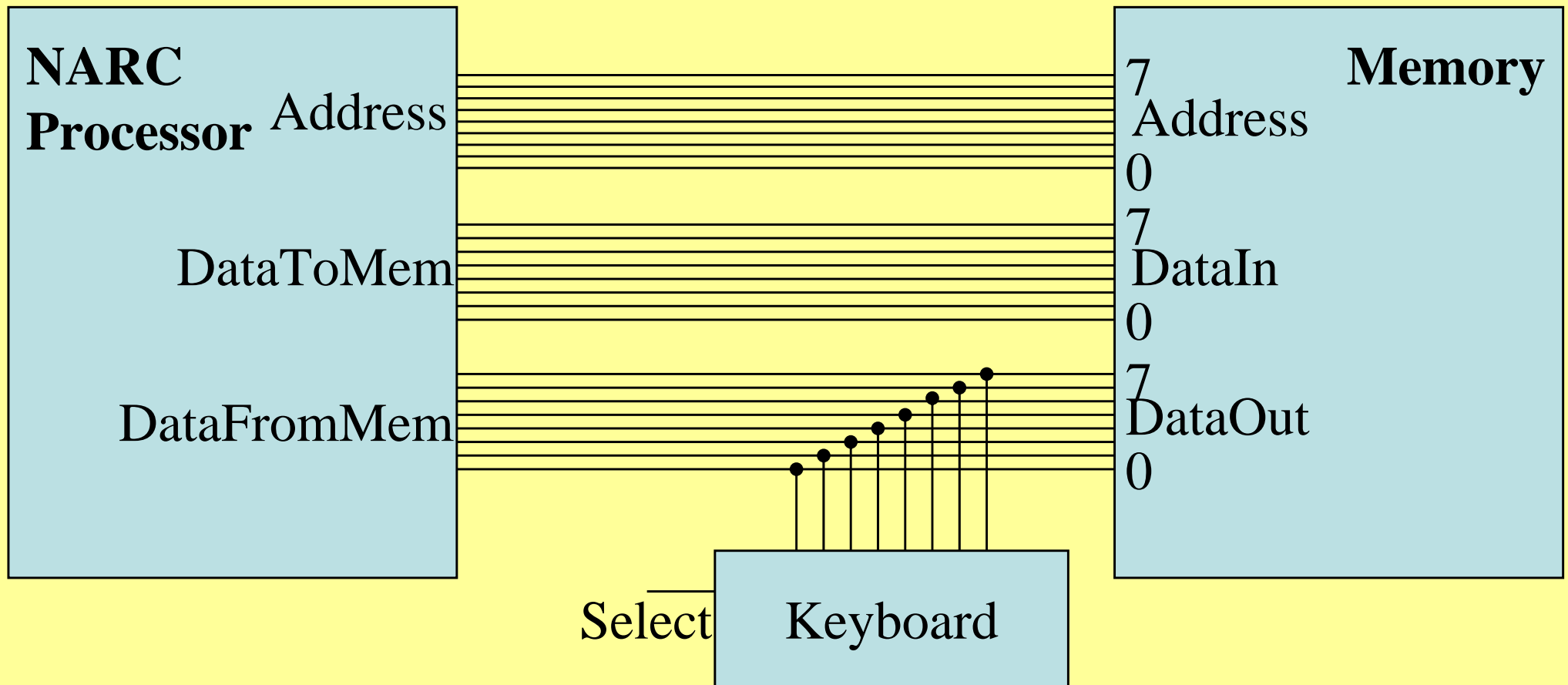
The processor is connected to the memory via three bundles of wire - for example, each 8 bits wide:



Idea: let's use the same wiring to connect the keyboard

Hitching a ride on the memory bus

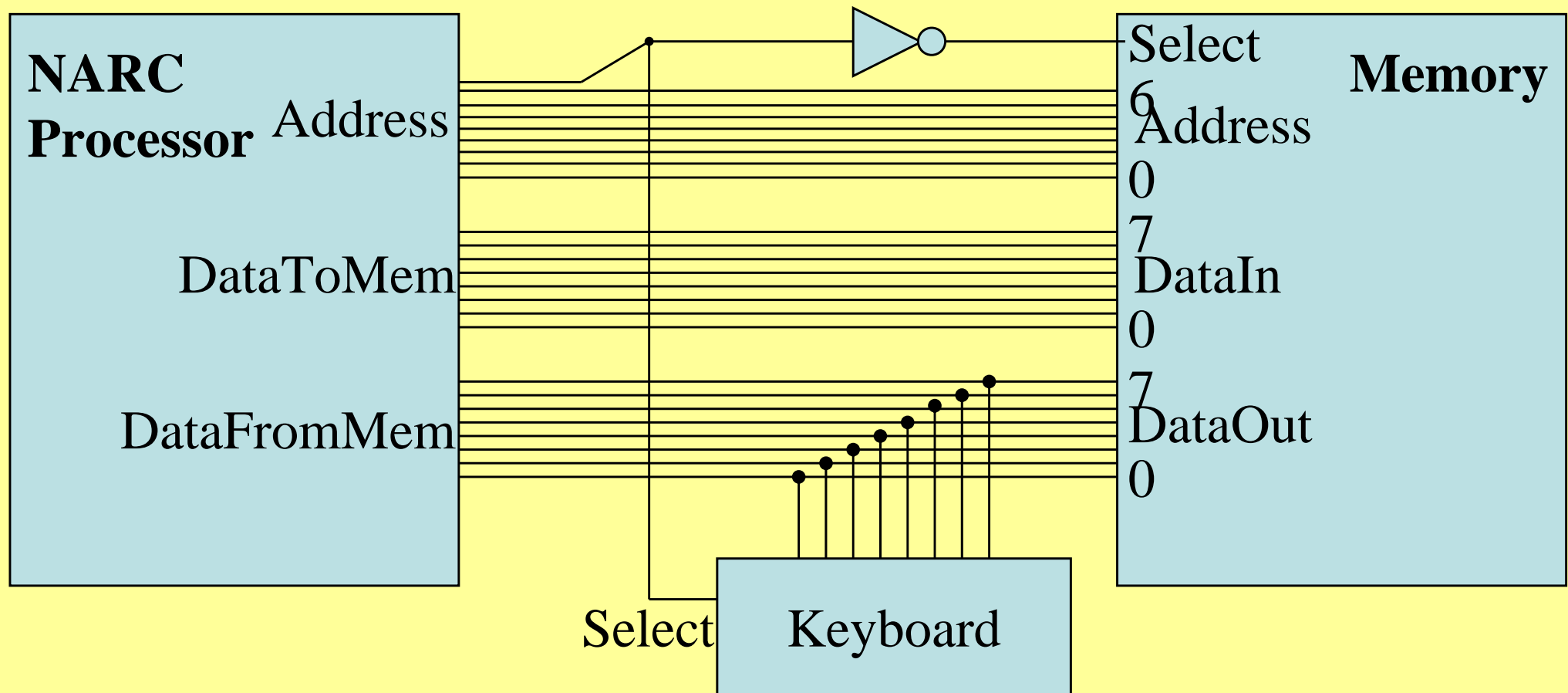
Let's attach the keyboard so it can send data to the processor just like the memory can:



“Select” signal controls whether the Keyboard signals are sent

"Memory-mapped" input/output

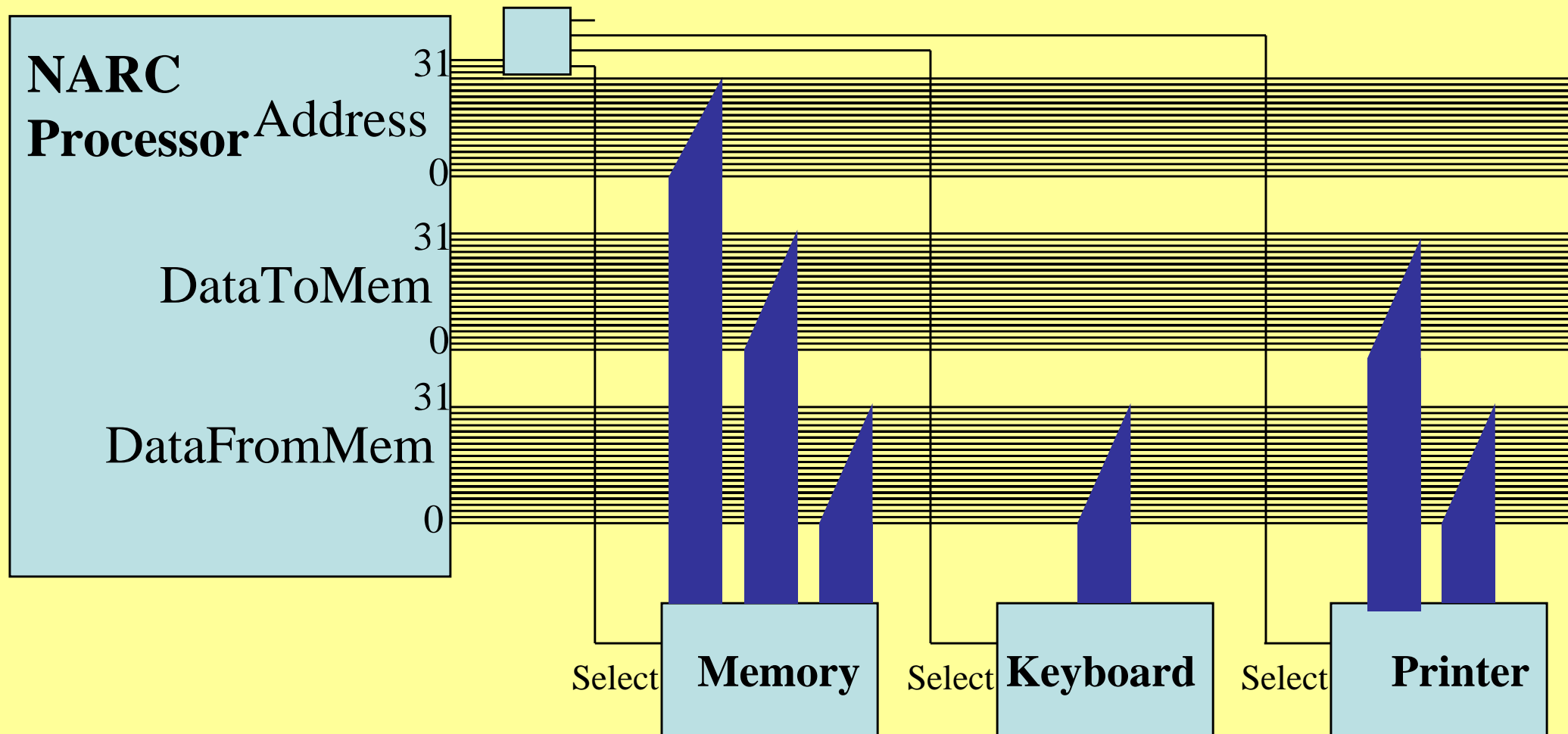
How can we get the processor to Select the keyboard?
Use one of the bits of the Memory address



Now we can sense the keyboard using a 'loadm' instruction

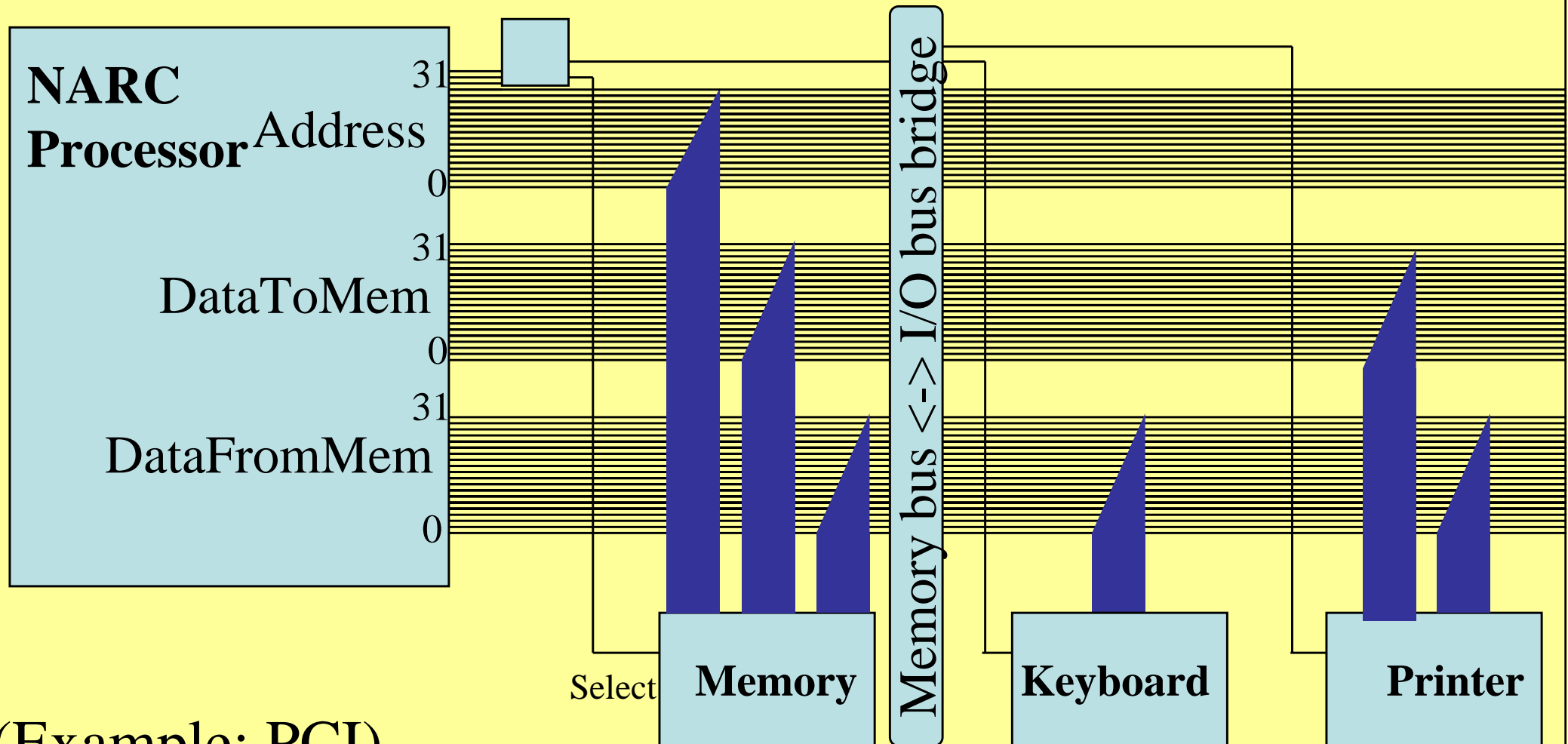
Connecting several devices

How can we get the processor to Select the keyboard?
Use one of the bits of the Memory address



Separate bus for input/output devices

- CPU-memory interface runs at very high speed
- Devices are connected to a separate, more robust bus



(Example: PCI)

Attaching devices to NARC - Summary

- In the NARC design shown here, a program communicates with input and output devices using load and store instructions
- Whether the load/store refers to memory, the keyboard or the printer depends on the address
- This is called “memory-mapped I/O”
- Some processors have “in” and “out” instructions
- Exactly like load and store, except processor generates a signal to indicate whether each bus request is for memory, or for I/O

Chapter 2 - Summary

- Ch.2 gives simplified view of how a computer works
- Should now understand how instructions are encoded, and how a “universal” machine can be built which can perform *any* computation
- Picture is incomplete - in later lectures we will see how real machines need one or two key features which are missing from the NARC presented here
- In particular, interrupts, address translation, privileged execution mode
- Real machines have further features for performance reasons - pipelining, registers, cache