

Operating Systems Concepts

Chapter 2: Computer Organisation
Simple computer and instruction set
Communicating with devices
Interrupts

Chapter 2 – Computer Organisation

- How does a computer work
 - How instructions are executed
 - How processor send /receive data from device

Operating System Objectives and functions

- Convenience – use/programming
- Efficiency – resource management
- Extensibility/Evolve(ability) – permit effective development, testing, and introduction of new system functions without interfering with the service
 - Hardware upgrades/new types of hardware
 - New services
 - Fixes – fix = new fault?

The OS as a resource manager

- Operating system is unique in that it controls the computer **from within** (normally control systems are external to the thing they control)
 - E.g. Thermostat and heater
- That is, OS is ordinary piece of software executed by the processor
- **Relinquishes control and must depend on the processor to allow it to regain control**
 - Passes control over so that processor can run application
- OS consists of a nucleus or **Kernel** which contains most frequently used functions

Evolution of OS

- Serial Processing (**stone age**)
 - Late 40's-mid 50's – programmer worked with hardware i.e. no OS
 - Run from console, light display toggle switches, machine code programs
 - **Scheduling** – signup sheet to reserve machine time
 - **Set up time** – job involved loading compiler_program saving and loading/linking

Evolution of OS cont

- Batch Processing (mid 60's IBSYS for the IBM 7070/7094 machines)
 - Improve
 - **monitor** (early OS) user not use machine/ person batches jobs together to run.
 - Picked up by monitor, run, branch back to monitor when complete

Evolution of OS cont.

- Multi-processor batch systems (Iron Age 60-70's)
 - Still had idle processor, IO is slow compared with processor
 - Memory had Monitor+ 2 applications then when one in IO other can use CPU
 - Multiprogramming or multi-tasking
 - This requires [] and []

Evolution of OS cont

- Time-sharing
 - Like multi-processor but **interactive**– e.g. transaction processing
 - OS interleaves CPU time among interactive terminals
- Protection?

Operating System Structure

- Requirements = more features added to OS = more lines of code
- Problems : chronically late delivery, bugs, performance
- Addressed by looking at software structure – modularity
- Large OS could be 10s millions lines of code! - > abstraction or hierarchical layers
 - Each level relying on lower to perform primitive functions

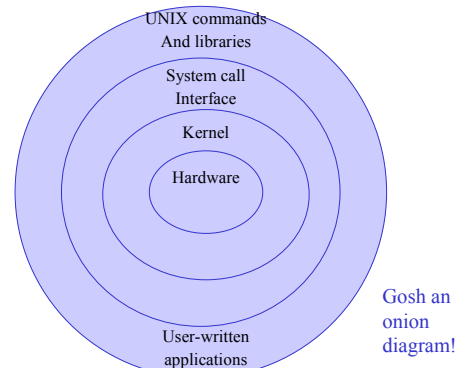
Hypothetical OS Hierarchy

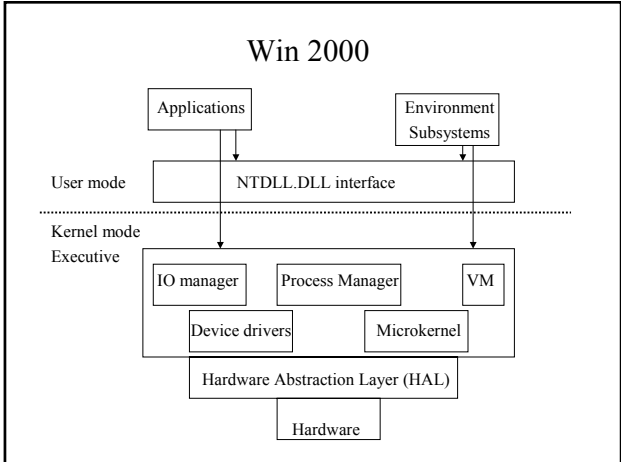
Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

Compare Win 2000 with UNIX

- **Unix**
 - Began 1970 bell labs
 - Rewritten in C (not assembly lang)
 - System call (crosses boundaries between user and kernel)
 - Not extensible
- **Win 2000**
 - MS-DOS for IBM pc (it had a disk!) 1981
 - Introduction of 80486/Pentium => windows OS
 - I.e. windows wasn't an OS until then
 - What was the OS called ?

UNIX (old)

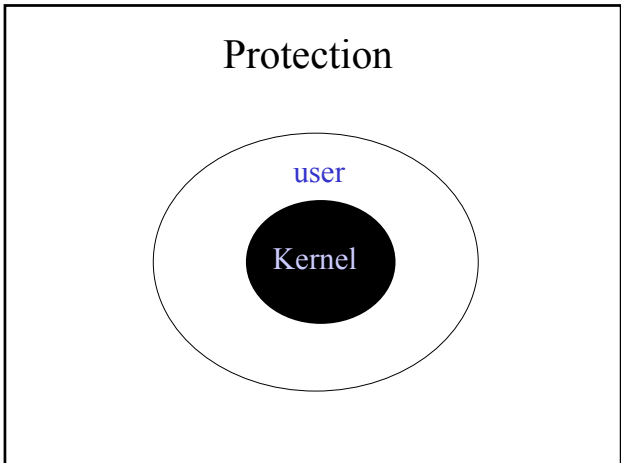




- ### Unix (modern)
- Modular structure
 - Eg module add x
 - Dynamically links in a module → **extensibility**

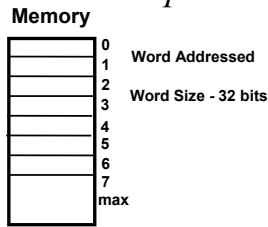
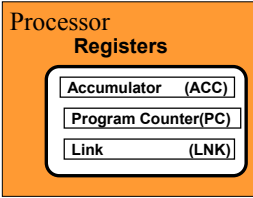
- ### Characteristics of Modern OS
- Until recently OS were **monolithic** kernels
 - Single process
 - Access to same address space
 - **Microkernel**
 - Essential functions in kernel (address space, scheduling, Inter processor Communication)
 - Other functions are processes (servers) run in user mode like an application

- ### Modern OS
- **Multithreading**
 - Process divided into threads running concurrently
 - Thread – dispatchable unit of work (program counter + stack pointer), interruptible, lightweight switching
 - Symmetric multiprocessing – parallelism
 - Distributed OS – looks like single filesystem, **SASOS**
 - **OOOS** – extensible, customisable, ease of development
 - **Component-based OS** – Go!

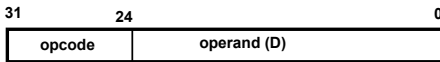


NARC not a real computer

NARC - not a real computer



- Processor operates on data items - “words” stored in memory
- Each word consists of 32 binary digits - “bits”
- Word may represent a number (or characters or anything...)
- Word may represent an instruction for the processor
- Processor is a machine which interprets (“executes”) instructions:



The NARC instruction set

Instruction	opcode	Meaning
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() {
    // This C function describes what the processor does to fetch an instruction
    int W;
    W = Mem[PC];
    Op = Opcode(W); // most significant 8 bits of W
    D = Operand(W); // least significant 24 bits of W
}

```

Execute

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

```

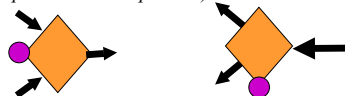
This C function describes what the processor does to execute an instruction

The “Fetch-execute cycle”

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

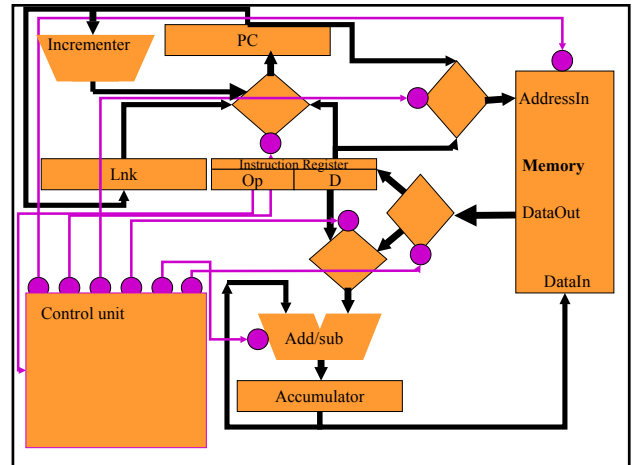
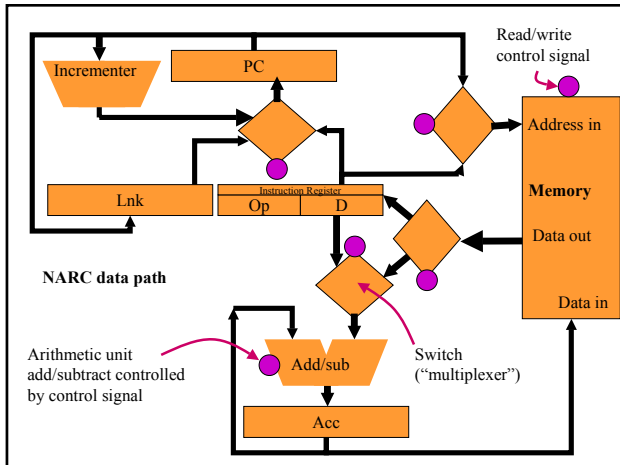
- Execution starts at location zero
- Instructions are stored in memory
- So are the computation’s working variables
- Fetch and execute are realised as a digital circuit

- 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**

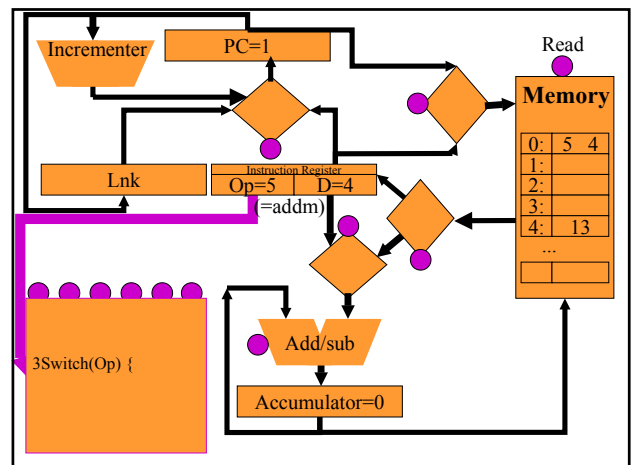
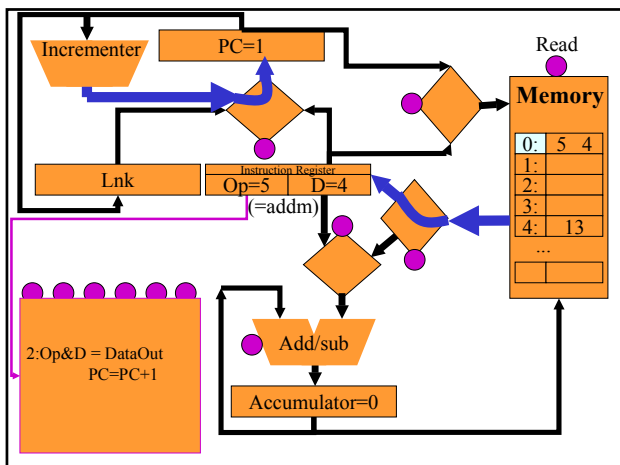
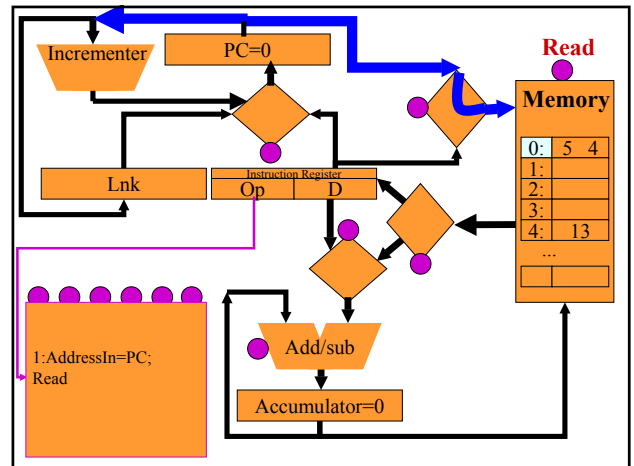
• In each case, what happens to the data is determined by a control signal

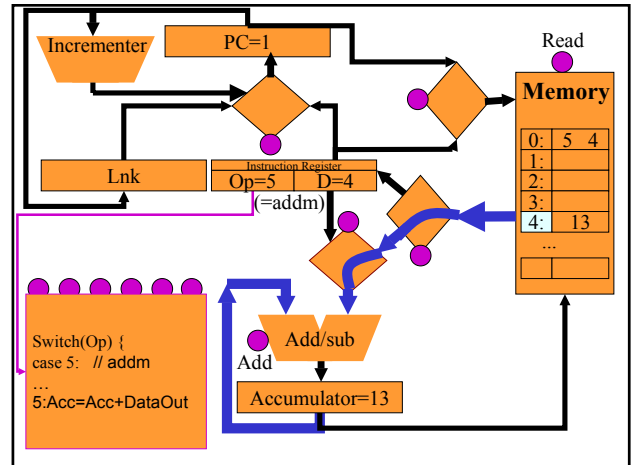
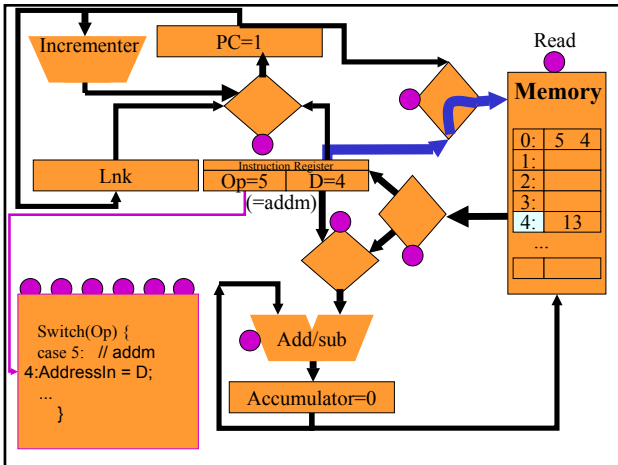


Instruction execution

- 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" ACC=ACC+memory[D] ...





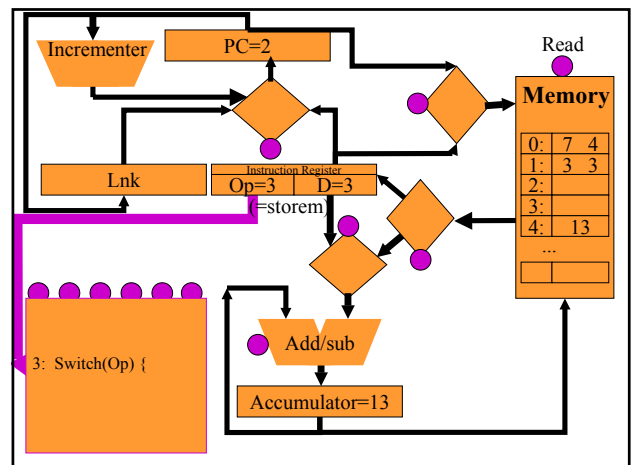
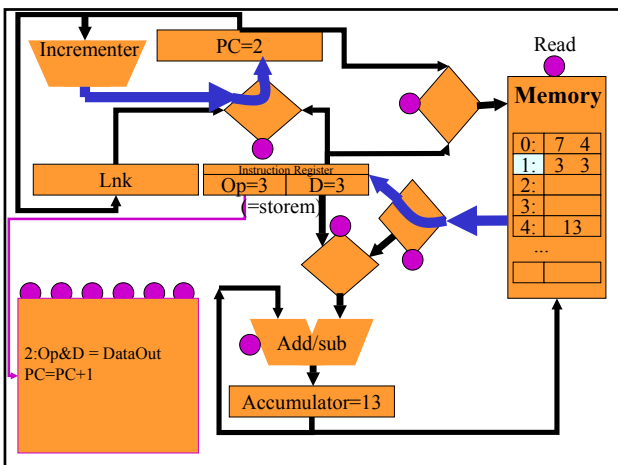
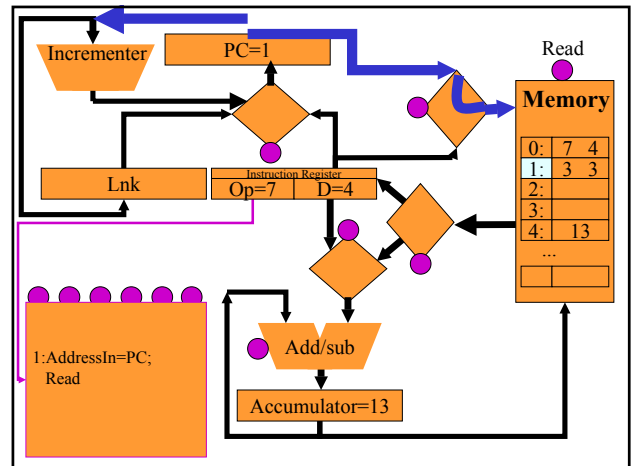
Exercise

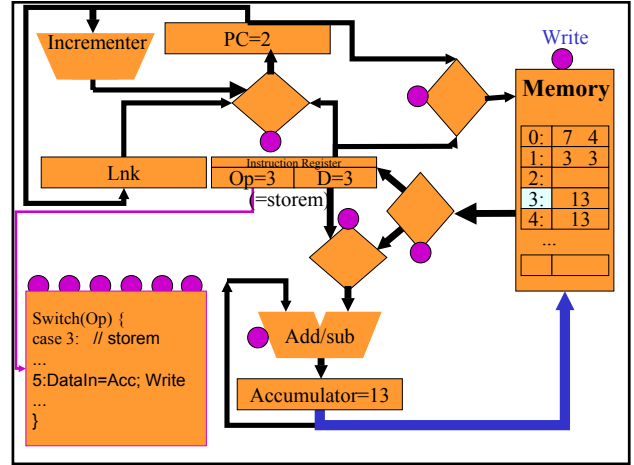
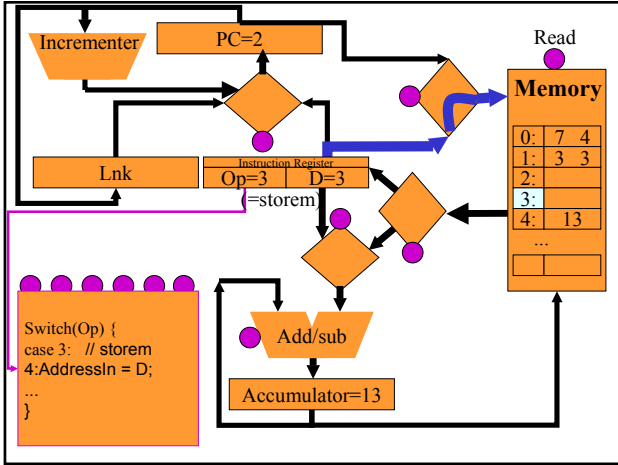
- Suppose that word 2 of the memory contains a "storem 3" instruction:

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

case 3: Mem(D) := Acc; // storem

- Trace through the sequence of operations to execute this instruction

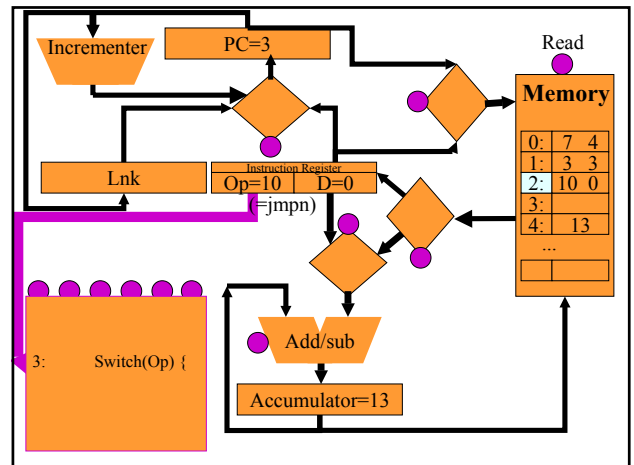
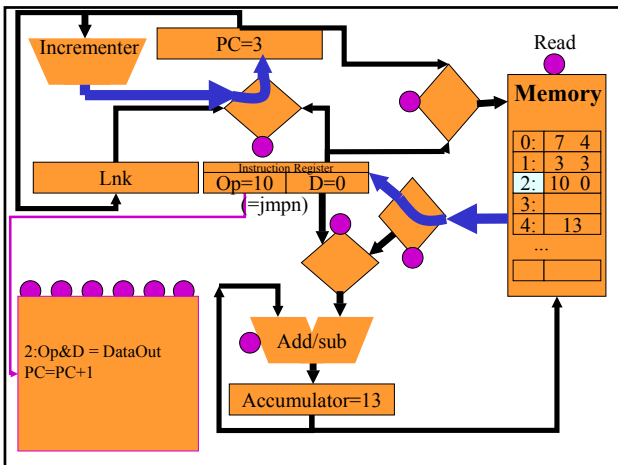
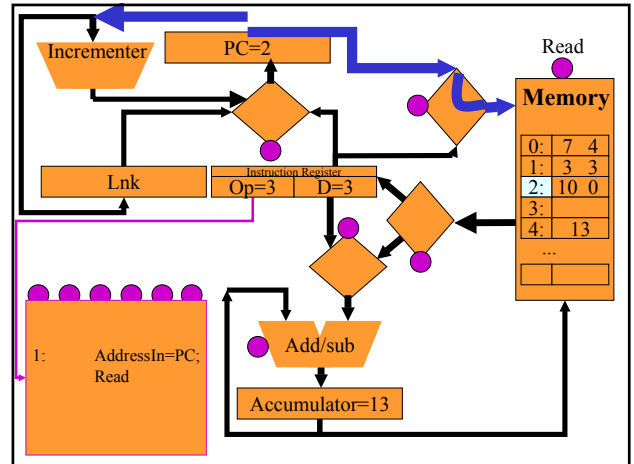


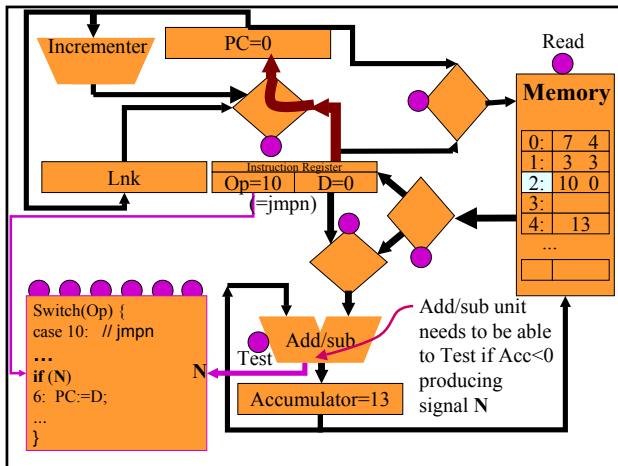
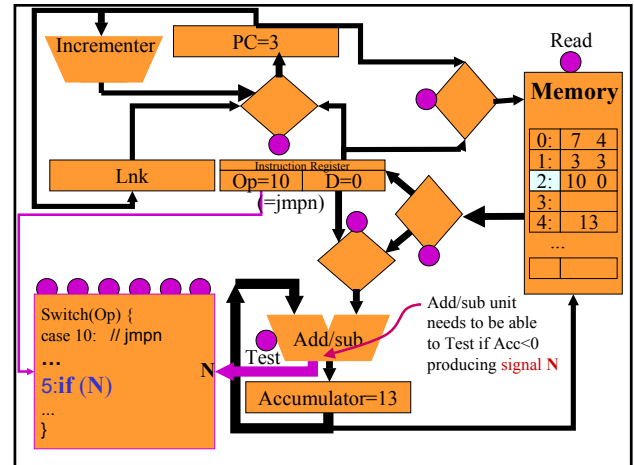
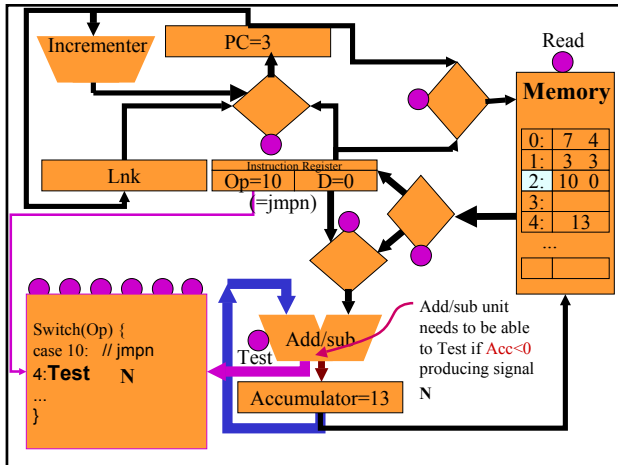


Exercise

- Suppose that word 3 of the memory contains "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!

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





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

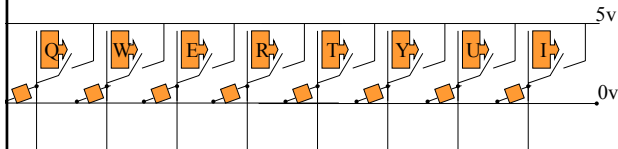
- Each instruction in the NARC instruction set is implemented by a sequence of steps in which data is moved around the data path
- E.g.. `addm D`:
 - 1: `AddressIn=PC; Read`
 - 2: `Op&D = DataOut; PC=PC+1`
 - 3: `Switch(Op) {`
 - 4: `AddressIn = D;`
 - 5: `Acc=Acc+DataOut`
- These steps are sometimes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

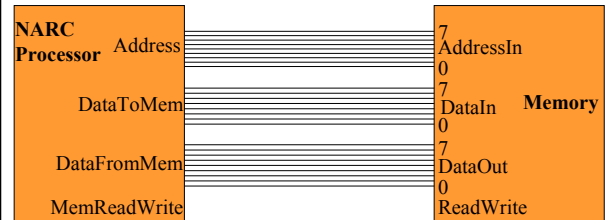
- Suppose we need to connect a keyboard to the NARC
- Each key generates a signal - 1/0
- Each key makes an electrical contact



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

Connecting a keyboard... the memory bus

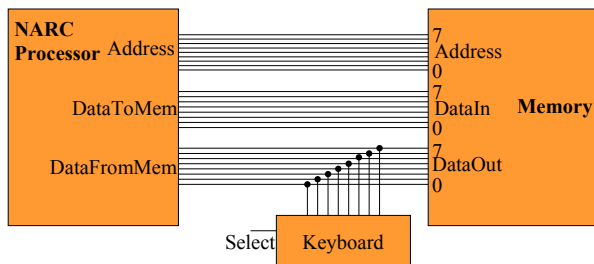
- The processor is connected to the memory via three bundles of wire - for example, each 8 bits wide:



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

Hitching a ride on the memory bus

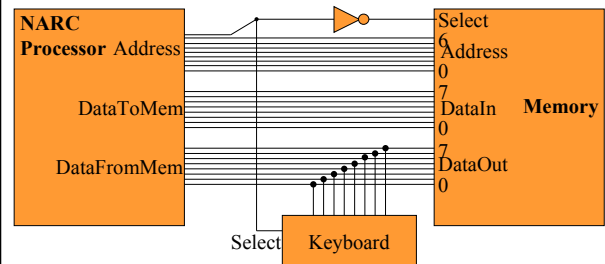
- Let's attach the keyboard so it can send data to the processor just like the memory can:



- "Select" signal controls whether the Keyboard signals are sent

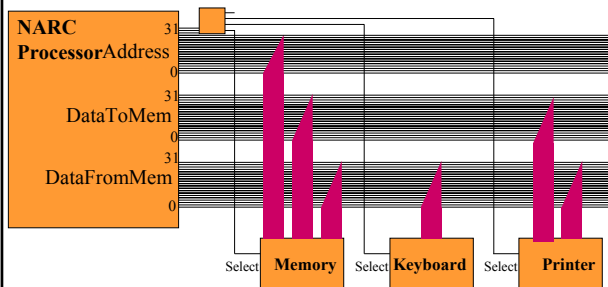
"Memory-mapped" input/output

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



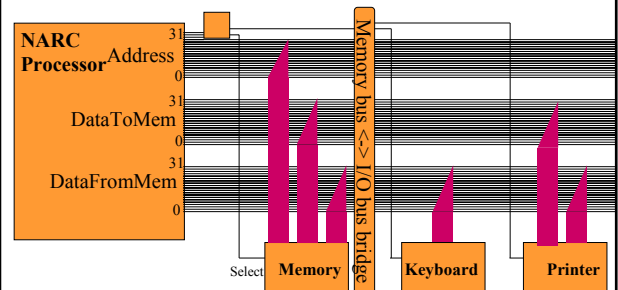
- Now we can sense the keyboard using a 'loadm' instruction

Connecting several devices



Separate bus for input/output devices

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



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 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**