von Neumann machine

• prior to von Neumann, special memories (e.g., mercury delay tubes, vacuum tubes) were used to store data, and programs were either stored in separate memories or physically wired-up on plugboards.

• von Neumann is credited with the idea of "stored program" computers, where the program is stored in the same memory as the data and indeed can be treated like data. (the idea originated with ENIAC pioneers Eckert and Mauchly)
von Neumann machine characteristics

- random-access, one-dimensional memory (vs. sequential memories)
- sequential processing (no pipelining)
- stored program, no hardware distinction between instructions and data vs "Harvard architecture" with separate instruction and data memories)
- binary, parallel by word, two's complement (vs. decimal, serial-by-digit, sign-magnitude)
- instruction fetch/execute cycle, branch by explicit change of PC (vs. following a link address from one instruction to the next)
- three-register arithmetic -- ACC, MQ, MBR
von Neumann machine characteristics

• memory contains series of bits grouped into addressable units

NOTE: binary devices are relatively easy to build – positive feedback drives the device to one of two extreme states and keeps it there, so the decimal storage devices of some old machines were usually four binary devices for which only ten of the sixteen states were used

• data is accessed in memory by naming memory addresses (like a big array)
Von Neumann machine characteristics

- addresses are consecutive binary integers (unsigned) - this is convenient for addressing arrays and sequentially stepping through a program

- bit strings have no inherent data type - can be integer, character string, machine instruction, or just garbage => HARDWARE DOESN'T KNOW DATA TYPE
Von Neumann machine

consider

```c
int main(){
    int i;
    char c = 'a';
    printf("%c 0x%x %d\n",c,c,c);
    i = c + 1;
    printf("%c 0x%x %d\n",i,i,i);
    return 0;
}
```

which prints

a 0x61 97
b 0x62 98

'a' is the (ASCII-encoded) bit string '01100001'; `printf()` can be told to format this bit string as a character, a hexadecimal number, or a decimal number; also `%s` can tell `printf()` to use a bit string as the address of a character string
note that in C programs, char parameters are passed as 32-bit words and printf() uses the lowest 8 bits for the %c format

```c
int main(){
    int i = ( 'a' << 8 ) | 'b';
    printf("%c 0x%x %d\n",i,i,i);
    i = i + 1;
    printf("%c 0x%x %d\n",i,i,i);
    return 0;
}
```

NOTE: a few old computers experimented with type fields ("tags") added to memory words to distinguish between data and instruction types. The key point is that programs can be manipulated as data (e.g., think of machine code generated/modified at run time; also compiler, linker, loader).
A compiler typically type-checks all the variables it uses so that all operations are legal

NOTES:

- a strongly-typed language allows few, if any, implicit type conversions
- in a strong and statically typed language, the compiler declares a compile-time error when types don't match (e.g., C++, Java)
- in a strong and dynamically typed language, the interpreter or other run-time system declares a run-time error when types don't match (e.g., Python)
- because C is generous in using implicit conversions, it is considered weakly-typed with static (compile-time) type checking
Varieties of processors

1. accumulator machine – as discussed earlier.
2. stack machine - operands are on stack, push from memory/pop into memory
3. load/store machine - computational instructions operate on general registers and load/store instructions are only ones that access memory
Varieties of processors

Accumulator Machine

CPU

ALU

ACC

PC

IR

Memory

ALU

Arithmetic Logic Unit

PC

Program Counter

MAR

Memory Address Register

MDR

Memory Data Register

IR

Instruction Register

ACC

Accumulator
Varieties of processors

Stack Machine

CPU

ALU

STACK

PC

IR

MAR

MDR

Memory

ALU

Program Counter

Memory Address Register

Memory Data Register

Instruction Register

Arithmetic Logic Unit
Varieties of processors

Load/Store Machine

CPU

ALU

REGISTER

FILE

PC

IR

MAR

MDR

Memory

ALU

Arithmetic Logic Unit

PC

Program Counter

MAR

Memory Address Register

MDR

Memory Data Register

IR

Instruction Register
Closer view of CPU - central processing unit

data path

- data registers such as ACC (accumulator) or set of general registers
- address registers such as SP (stack pointer) and Xn (index registers)
- ALU (arithmetic and logic unit) which executes operations
- internal buses (transfer paths)
memory bus interface / BIU (bus interface unit)

- MAR (memory address register) – a CPU register that either stores the memory address from which data will be fetched to the CPU or the address to which data will be sent and stored

- MDR (memory data register -- perhaps better called memory _buffer_ reg) – a CPU register that contains the data to be stored in memory or the data after a fetch from memory.
Closer view of CPU - central processing unit

control

- PC (program counter) - points to next instruction (x86 calls it the IP == instruction pointer)
- IR (instruction register) - holds current instruction
- PSR (processor status register) - indicates results of previous operation (called condition codes or flags)
closer view of CPU - central processing unit

PC (program counter) ↔ ...

IR (instruction register) ↔ ...

PSR (processor status register)

ACC

MAR (memory address register) → to and from memory

MDR (memory data register) ←

ALU

Move PC to MAR

read memory

Control signals to cause datapath actions are generated by logic in the control unit
fetch-execute cycle

- each instruction generates a sequence of control signals (instruction fetch is the same for each instruction)
- the control signals determine the register transfers in data path
- timing - how long should the control signals be active? data signals must travel from a register, across the bus, through the ALU, to a register => minimum clock cycle time
for A := B + C
assembly code / action on computer

; load(B) ; ACC <- memory[B]
;
add(C) ; ACC <- ACC + memory[C]
;
store(A) ; memory[A] <- ACC

like a switch statement – decode selects one of these execution paths

register is loaded with a copy of the value in memory[B]
value in ACC register is added with a copy of the value in memory[C], and sum is placed in ACC register
memory[A] gets a copy of the value in ACC register
register transfers (datapath actions) for these three instructions

---fetch---
MAR $\leftarrow$ PC
READ
IR $\leftarrow$ MDR
PC $\leftarrow$ PC + incr

---decode---
decode IR

---execute---
load(B)
MAR $\leftarrow$ addr(IP)
READ
ACC $\leftarrow$ MDR

add(C)
MAR $\leftarrow$ addr(IP)
READ
ACC $\leftarrow$ ACC + MDR

store(A)
MAR $\leftarrow$ addr(IP)
MDR $\leftarrow$ ACC

WRITE

can check for external signals (interrupts) between instructions